Titre: Title:	Nouvelle architecture pour les environnements de développement intégré et traçage de logiciel
Auteur: Author:	Yonni Chen Kuang Piao
Date:	2018
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Chen Kuang Piao, Y. (2018). Nouvelle architecture pour les environnements de développement intégré et traçage de logiciel [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/3282/

UNIVERSITÉ DE MONTRÉAL

NOUVELLE ARCHITECTURE POUR LES ENVIRONNEMENTS DE DÉVELOPPEMENT INTÉGRÉ ET TRAÇAGE DE LOGICIEL

YONNI CHEN KUANG PIAO
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2018

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

NOUVELLE ARCHITECTURE POUR LES ENVIRONNEMENTS DE DÉVELOPPEMENT INTÉGRÉ ET TRAÇAGE DE LOGICIEL

présenté par : CHEN KUANG PIAO Yonni
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. OZELL Benoit, Ph. D., président
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. GUIBAULT François, Ph. D., membre

DÉDICACE

À toute ma famille et à mes amis de Montréal,
vous me manquerez...

REMERCIEMENTS

Je ne pourrais commencer ce mémoire sans prendre le temps de remercier toutes les personnes exceptionnelles que j'ai rencontrées et qui ont contribué à la réussite de ma maîtrise.

Je tiens premièrement à remercier Michel Dagenais, mon directeur de recherche pour m'avoir confié ce projet tout en m'apportant soutien et conseils durant mes travaux. Sa disponibilité, son suivi et ses rétroactions m'ont permis d'atteindre mes objectifs les plus ambitieux.

Je souhaite également remercier les partenaires du laboratoire pour le financement de ce projet de recherche. Outre le financement, les équipes d'Éricsson, d'EfficiOS et Google ont démontré leur intérêt à la réussite du projet en s'impliquant dans nos discussions et nos rencontres, tout en m'aidant tout au long de mes travaux. Un remerciement tout particulier s'adresse à l'équipe de Trace Compass pour m'avoir accueilli en stage, pour m'avoir fait confiance et pour m'avoir poussé à toujours dépasser les attentes.

Finalement, merci à mes collègues du laboratoire DORSAL pour les encouragements, les rires et les gâteaux du vendredi après-midi. Au-delà des moments agréables passés, j'aimerais exprimer ma reconnaissance aux deux associés de recherche, Naser et Geneviève, pour m'avoir guidé tout au long de cette maîtrise et de m'avoir partagé leurs expériences sur le projet.

RÉSUMÉ

La conception et le développement de logiciels requièrent souvent l'utilisation d'un Environnement de Développement Intégré (EDI) pour assister et faciliter le travail des développeurs. Les EDI offrent, à travers une interface graphique, des outils pour l'édition, la compilation et le débogage du code. Cependant, lorsque ces outils ne sont pas adaptés et suffisants pour la détection de défauts de performance sur des logiciels complexes, comme les systèmes distribués, les développeurs se tournent vers des techniques de traçage. Des logiciels appelés traceurs récoltent des informations précises pendant l'exécution du système instrumenté, et les regroupent dans une trace. Une trace peut contenir une quantité importante de données. Des outils spécialisés ont été développés afin d'en automatiser le processus d'analyse et de visualisation. Au fur et à mesure qu'un logiciel grandit et se complexifie, l'utilisation de ces outils d'analyse et de visualisation devient tout aussi importante qu'un débogueur.

Néanmoins, ces outils sont complexes, autonomes et difficilement réutilisables dans d'autres systèmes. De plus, ils ne supportent pas les mêmes analyses, les mêmes formats de trace, ni les mêmes cas d'utilisation, ce qui implique que le développeur ait besoin d'installer plusieurs outils pour arriver à ses fins. Dans le cadre de ce projet, nous cherchons donc à résoudre ces problèmes et à intégrer l'analyse et la visualisation de trace non seulement dans les EDI, mais dans tout autre système qui pourrait en bénéficier, tels que les serveurs d'intégration continue ou encore les systèmes de monitorage. Par conséquent, nous proposons une nouvelle architecture logicielle flexible basée sur une approche client-serveur, d'architecture orientée service et multicouche.

Notre travail s'étend à l'implémentation de l'architecture du serveur au sein du projet Trace Compass et l'implémentation de l'architecture du client au sein d'un nouveau projet appelé TraceScape. Toutes nos contributions sont disponibles à code source ouvert. Des tests de performance ont été menés afin d'évaluer le surcoût associé à la nouvelle architecture par rapport à la précédente approche, et les résultats indiquent un surcoût acceptable.

ABSTRACT

Creating software often requires using an Integrated Development Environment (IDE) to help and facilitate the development work. With a simplified user interface, IDEs provide many useful tools such as a code editor, a compiler, and a debugger. Nonetheless, when those tools are not enough to detect performance defects in a large, complex and multithreaded system, developers use tracing techniques. A program called tracer collects accurate information during the execution of an instrumented system. A trace could contain a lot of data, and specialized tools have been developed to analyze traces automatically and show the results in interactive views. As the software grows and becomes more complex, using trace visualization tools must be part of the developer tool environment, like the debugger in the software development process.

However, trace visualization tools are sophisticated, standalone and hardly reusable in other systems such as an IDE. Moreover, they have their specific trace format support, specific use cases, and specific trace analyses. Most of the time, developers need to install and use several such tools to fulfill their needs. In this research project, we aim to solve those problems and integrate trace analysis and visualization in tools such as IDEs, monitoring systems or continuous integration systems. Thus, we propose a flexible software architecture based on client-server, service-oriented architecture and layered approaches.

We implemented the server architecture in the Trace Compass project and the client architecture in a new project called TraceScape. All of our contributions are available online in open source repositories. We also evaluated our proposed architecture through benchmarks, and the results show that our approach has an acceptable overhead compared to the standalone approach.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

API	Application Interface
BSON	Binary JSON
CPU	Central Processing Unit
CTF	Common Trace Format
EDI	Environnements de Développement Intégré
IDL	Interface Definition Language
JSON	JavaScript Object Notation
LSP	Language Server Protocol
LTTng	Linux Trace Toolkit next generation
REST	Representational State Transfer
RPC	Remote Procedure Call
SHT	State History Tree
SOAP	Simple Object Access Protocol
TASP	Trace Analysis Server Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

La conception et le développement de logiciels requièrent très souvent l'utilisation d'un Environnement de Développement Intégré (EDI) afin de faciliter la tâche aux développeurs. Les EDI regroupent des outils indépendants et complexes tels qu'un éditeur de code, un compilateur et un débogueur. Néanmoins, lorsque ces outils ne sont pas suffisants ou adéquats pour détecter des défauts de performance, les développeurs se tournent vers des techniques de traçage afin de récolter des données pendant l'exécution du logiciel. Ces données, agrégées et écrites dans un fichier appelé trace, peuvent atteindre l'ordre de dizaines de gigaoctet sur disque. Pour éviter de devoir analyser manuellement les traces, des outils automatiques de visualisation de trace ont été conçus pour assister le développeur.

Malheureusement, ces outils sont souvent autonomes et n'interagissent pas avec l'EDI, perdant ainsi des fonctionnalités intéressantes fournies par ce dernier. De plus, les outils ne supportent pas tous les mêmes formats de trace, les mêmes analyses et les mêmes cas d'utilisation. Cela oblige le développeur à installer et utiliser différents outils. L'ensemble des recherches et travaux décrits dans ce mémoire vise à répondre à ce problème en proposant une nouvelle architecture. L'objectif final de cette architecture est d'intégrer l'analyse et la visualisation de n'importe quelle trace au sein des EDI.

1.1 Définitions et concepts de base

1.1.1 Environnement de Développement Intégré

Un Environnement de Développement Intégré (EDI) est une application regroupant plusieurs outils pour assister le développeur dans les tâches reliées au développement et à la conception de logiciels. À travers une interface graphique, un EDI met à la disposition de l'utilisateur des outils permettant entre autres d'écrire du code, compiler et déboguer le code, tester des fonctionnalités et gérer les versions du code. L'objectif de ces outils est d'automatiser et simplifier les opérations du processus de développement, ce qui augmente la productivité du développeur. Ces outils sont soit intégrés dès le début, donc spécialement conçus pour fonctionner en synergie, ou soit développés indépendamment et assemblés *a posteriori*.

Parmi les EDI les plus populaires, nous avons Microsoft Visual Studio, Eclipse, Apple XCode et Jetbrains IntelliJ.

1.1.2 Traçage

Le traçage est une technique d'analyse de systèmes informatiques qui consiste à enregistrer des évènements pendant l'exécution de ces derniers. Avant de collecter les évènements, des points de trace doivent être insérés de façon statique ou dynamique dans le système. Dans le premier cas, les points de trace sont insérés manuellement dans le code source puis le programme est recompilé. Dans le second cas, les points de traces sont ajoutés dynamiquement pendant l'exécution du programme. Lorsque le système exécute la ligne de code contenant un point de trace, un évènement est émis et un logiciel indépendant appelé traceur le consomme pour éventuellement l'écrire dans un fichier appelé trace. Un évènement décrit une action (appel système, appel de fonction, etc.) qui s'est produite dans le système avec une estampille de temps. Des informations sur le contexte d'exécution de l'action sont également enregistrées. Le contenu de la trace est organisé selon un certain format. Une trace peut facilement contenir des millions, voire des milliards d'évènements. Analyser manuellement ces évènements, un après l'autre en ouvrant le fichier de trace, devient donc une tâche impossible. Pour répondre à ce problème, des outils automatiques de visualisation de traces ont été développés et viennent assister l'utilisateur en offrant des vues interactives sur les résultats d'analyse de trace.

1.1.3 Architecture logicielle

L'architecture logicielle est une description abstraite de la structure et de l'organisation d'un ou de plusieurs systèmes. Un système est une collection de composantes logicielles et les interactions entre ces composantes sont appelées connecteurs. À l'aide de schémas, l'architecture logicielle expose comment un système doit être conçu afin de répondre à des exigences techniques et fonctionnelles, tout en prenant en considération d'autres qualités logicielles telles que la performance, la réutilisabilité ou la flexibilité.

1.1.4 Système distribué

Un système distribué est une collection de composantes logicielles indépendantes qui communiquent à travers des messages. Ces composantes, déployées sur des machines pouvant être à des emplacements géographiques différents, apparaissent comme un seul système cohérent pour ses utilisateurs. Les composantes logicielles peuvent également être conçues avec des langages de programmation différents.

1.1.5 Infonuagique

L'infonuagique consiste à accéder et à exploiter des ressources informatiques situées sur des machines distantes par l'intermédiaire d'un réseau. Les ressources, fournissant une puissance de calcul ou de stockage, sont utilisées et facturées à la demande. L'infonuagique offre trois types de services : l'infrastructure en tant que service (*Infrastructure as a Service* (IaaS)), la plateforme en tant que service (*Platform as a Service* (PaaS)) et le logiciel en tant que service (*Software as a Service* (SaaS)).

1.2 Éléments de la problématique

Traditionnellement, les EDI supportaient plusieurs langages de programmation par le biais de modules d'extensions (*plugin*). Malheureusement, ces modules étaient fortement couplés à l'EDI, ce qui les rendaient difficilement réutilisables. Ainsi, pour m EDI qui supportent chacun n langages de programmation, les développeurs devaient écrire et maintenir $m \times n$ modules. Par ailleurs, des langages de programmation comme le C++ ont une syntaxe complexe et développer un compilateur ou un débogueur de haute qualité requiert une quantité significative d'efforts et de ressources. Il serait inefficace de tout réécrire à chaque fois. Ces problèmes ont motivé les récents travaux sur les EDI à réduire le nombre de modules à développer, le ramenant à $m + n$, et favoriser la réutilisabilité.

Cependant, lorsque les outils fournis par les EDI ne suffisent pas pour détecter des défauts de performance sur de gigantesque systèmes multicoeurs, les développeurs se tournent vers des techniques d'analyse dynamique telles que le traçage. Les traceurs collectent des données durant l'exécution du programme, et regroupent et organisent toutes ces informations dans une trace. Des programmes en ligne de commande tels que `babeltrace` sont pratiques pour la lecture des traces au format binaire CTF mais deviennent rapidement insuffisants. D'autres outils spécialisés et plus sophistiqués assistent les développeurs dans l'analyse automatique et la visualisation de ces données. Au fur et à mesure qu'un système grandit et se complexifie, ces outils deviennent indispensables, au même titre qu'un débogueur, dans le processus de développement logiciel. Malheureusement, ces derniers sont souvent autonomes et n'interagissent pas avec les autres outils de l'EDI. En effet, une fois le défaut trouvé dans l'outil d'analyse, il serait pertinent de savoir quelles procédures ou lignes de code en sont responsables, afin de les corriger. Par ailleurs, chaque outil de visualisation de trace n'offre pas les mêmes analyses, ne supporte pas les mêmes formats de trace et n'est pas destiné au même cas d'utilisation. Il en résulte que le développeur est obligé d'installer plusieurs outils indépendants et autonomes sur sa machine pour répondre à ses besoins. Une solution archi-

tecturale permettant d'intégrer l'analyse et la visualisation de traces directement au sein de n'importe quel EDI devient donc une nécessité.

1.3 Objectifs de recherche

Les problématiques énoncées à la section précédente font ressortir la question de recherche suivante : comment peut-on modifier et adapter les outils d'analyse et de visualisation de trace existants afin qu'ils soient plus facilement utilisables et intégrés dans les EDI ?

Il en découle les objectifs de recherche suivants :

1. Analyser le fonctionnement et l'architecture de Trace Compass.
2. Proposer une nouvelle architecture logicielle flexible pour l'analyse et la visualisation de trace.
3. Procéder à l'implémentation de la solution proposée et à la réingénierie de Trace Compass.
4. Valider que la solution n'ajoute qu'un surcoût acceptable et n'affecte pas les fonctionnalités déjà présentes.

1.4 Plan du mémoire

Le chapitre 2 présente l'état de l'art des solutions de traçage, de l'analyse de trace et de l'architecture logicielle, tout en mettant l'accent sur les récents travaux reliés aux EDI. Il présentera également des outils de visualisation de trace tout en décrivant des techniques pour l'interopérabilité des systèmes hétérogènes. Ensuite, le chapitre 3 décrira la méthodologie utilisée pour obtenir les résultats. Le chapitre 4 expose un article de revue présentant en détail la solution et les résultats obtenus. Avant de conclure au chapitre 6, une discussion générale sera amorcée au chapitre 5.

CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre présente l'état de l'art du traçage, de l'analyse de trace et de l'architecture logicielle, tout en mettant l'accent sur les récents travaux reliés aux EDI. De plus, des solutions existantes pour la visualisation de traces seront exposées et une dernière section décrira l'interopérabilité des systèmes hétérogènes.

2.1 Traçage

Le traçage peut intervenir à plusieurs niveaux d'un système, que ce soit en espace noyau, en espace utilisateur, au niveau de la machine virtuelle et de l'hyperviseur ou encore au niveau du réseau. Cette section présente les solutions existantes dans le domaine du traçage pour les différentes plateformes.

2.1.1 Outils de traçage sur les systèmes Linux

`strace` est un outil permettant de tracer les appels systèmes effectués par un processus durant son exécution (Desnoyers and Dagenais, 2006). Son utilisation ne nécessite pas de recompilation ni de reconfiguration du code source de l'application. Cet outil est basé sur l'appel système `ptrace` et il permet également de surveiller les signaux et les changements d'état des processus (Gregg, 2014). Le format du contenu des traces est textuel et lisible par l'utilisateur. Cet outil est peu flexible car l'ensemble d'événements qu'il est possible de tracer est fixe. En outre, le surcoût du traçage est très important car plusieurs changements de contexte entre l'application et `strace` sont requis à chaque appel système.

`ftrace` est un outil de traçage, intégré au noyau Linux, permettant de mesurer les temps d'exécution des fonctions du noyau (Bird, 2009). Il est également possible de collecter les informations sur les interruptions et les évènements d'ordonnancement, afin de détecter des problèmes de latence (Rostedt, 2009). Les fichiers de configurations et les traces produites de `ftrace` se trouvent sur le système de fichier `debugfs`. Le format du contenu des traces est textuel et lisible par l'utilisateur.

`Perf` est un outil d'analyse de performance intégré au noyau qui permet de tracer les compteurs de performance matériels et logiciels (Edge, 2009). Les compteurs mesurent le nombre d'instructions exécutées, les fautes de cache, les fautes de page ou encore les changements de contexte. Les informations recueillies peuvent soit être écrites dans un fichier au format textuel ou soit être directement affichées sur la console sous forme de statistiques.

`Linux Trace Toolkit next generation` (LTTng) est un traceur à code source ouvert développé à Polytechnique Montréal (Desnoyers and R Dagenais, 2006). Conçu pour avoir le plus faible surcoût d'utilisation, il permet non seulement de tracer le noyau, mais également les programmes en espaces utilisateurs avec le module LTTng-UST. LTTng utilise des tampons circulaires dans lesquels les évènements sont d'abord insérés avant d'être consommés et écrits sur disque. Le format du contenu des traces est le *Common Trace Format* (CTF) qui est binaire. LTTng permet de récolter des informations similaires à `ftrace` et `Perf`, avec un surcoût légèrement plus faible (Gebai and Dagenais, 2018), mais offre des fonctionallités supplémentaires comme des sessions concurrentes, le traçage en mode utilisateur ou la rotation des fichiers de trace.

`SystemTap` est un traceur développé par Red Hat dont la conception et le fonctionnement sont similaires à `DTrace` (Prasad et al., 2005). Il supporte le traçage en espace noyau et en espace utilisateur par le biais de scripts au langage dédié et qu'il faut attacher à des points de trace. L'insertion des points de trace peut être soit statique ou dynamique. Le contenu des traces est directement affiché sur la console ou écrit en format textuel dans un fichier.

2.1.2 Outils de traçage sur les systèmes Windows

`Event Tracing for Windows` (ETW) est le système de traçage développé par Microsoft pour les plateformes Windows (Park and Buch, 2007). L'outil permet de tracer le noyau et l'espace utilisateur. Un mécanisme de tampons par CPU, similaire à LTTng, est utilisé pour insérer les évènements avant l'écriture sur disque de la trace. De plus, afin de faciliter l'analyse de trace, l'outil prend un cliché de l'état du système avant la session de traçage. Le format du contenu des traces est binaire.

2.1.3 Outils de traçage sur les systèmes de calcul haute performance

`Vampir` est une suite logicielle commerciale destinée à l'analyse de performance des systèmes de calcul haute performance (*High Performance Computing* (HPC)). Le module `VampirTrace` trace et instrumente les programmes séquentiels, les programmes utilisant MPI, OpenMP, OpenCL ou encore CUDA (Knüpfer et al., 2012). Les informations sur les compteurs de performance, les entrées-sorties et l'utilisation de la mémoire sont recueillies et les traces produites sont au format OTF (*Open Trace Format*) qui est textuel ou au format OTF2 qui est binaire (Eschweiler et al., 2011).

`MPI Parallel Environment` (MPE) est un ensemble d'outils pour instrumenter des programmes utilisant MPI (*Message Passing Interface*) et qui s'exécutent sur plusieurs machines

(Wu et al., 2000). L'ensemble comprend entre autres des profileurs, des débogueurs et des visualiseurs. Les temps passés dans les routines MPI et les informations relatives à ces appels sont enregistrés dans un journal d'historique (*log*). Le format de ces fichiers est soit le ALOG, le CLOG ou encore le SLOG.

`Open SpeedShop` est un ensemble d'outils multiplateforme pour l'analyse de performance des applications HPC qui s'exécutent sur les plateformes Intel, AMD, ARM, PowerPC, Cray et IBM (Schulz et al., 2008). L'outil regroupe des informations sur les compteurs de performance, les entrées-sorties et l'utilisation de la mémoire. Il trace et profile également les appels de fonctions MPI, OpenMP et CUDA. Les traces produites sont au format OTF.

2.1.4 Outils de traçage sur les systèmes distribués

`Magpie` est le système de traçage de Microsoft pour les systèmes distribués s'exécutant sur les plateformes Windows (Barham et al., 2003). Le système utilise `ETW`, l'API du profileur .NET, Detours et `tcpdump` pour récolter des informations sur les appels systèmes, les activités du noyau, les entrées-sorties et les communications réseaux. Magpie a été conçu pour l'instrumentation des services web.

`Dapper` est l'infrastructure de traçage de systèmes distribués de Google (Sigelman et al., 2010). Son design et sa conception sont fortement inspirés de `Magpie` et `X-Trace` mais limitent son instrumentation à quelques librairies. Dapper trace récursivement les requêtes utilisant des *Remote Procedure Calls* (RPC) et leur assigne un numéro d'identifiant unique. Chaque évènement est alors associé à un numéro d'identifiant unique, ce qui permet de regrouper par la suite les évènements par requête.

`OpenTracing`[1], inspiré de `Dapper`, est un projet de la *Cloud Native Computing Foundation* dont l'objectif est de standardiser l'instrumentation des systèmes distribués pour le traçage. Dans OpenTracing, le concept de trace permet de suivre l'état d'une transaction ou une requête à travers le système. Pour ce faire, ce dernier fournit un API et décrit une trace comme étant un graphe orienté acyclique de *spans*. Un *span* est un intervalle de temps qui représente le temps écoulé pour l'aboutissement d'une action. Finalement, OpenTracing étant une spécification, les développeurs peuvent ajouter ou changer très facilement d'implémentation.

2.2 Analyse et traitement de trace

L'analyse de trace est très souvent une tâche effectuée *a posteriori*. Séparer le traçage de l'analyse en deux opérations distinctes permet d'avoir le plus faible surcoût possible sur le

1. http ://opentracing.io/

système, tout en ayant des analyses plus coûteuses, mais également plus précises. Une fois l'analyse complétée, les résultats sont affichés dans un outil de visualisation de trace. Cette section présente différentes méthodes pour l'analyse de trace avant de décrire les solutions existantes pour la visualisation.

2.2.1 Gestionnaire d'état

La méthode du gestionnaire d'état consiste à bâtir une machine à états à partir des évènements de la trace (Wininger, 2014). Cette machine est créée lorsque la trace est lue depuis le début jusqu'à la fin, sans retour en arrière. Initialement, le gestionnaire contient des états vides jusqu'à ce que des évènements pertinents soient lus pour entraîner un changement d'état. La figure 2.1, inspirée des travaux de Wininger, illustre l'évolution des états d'un fil d'exécution à travers le temps depuis son initialisation en fonction des évènements de la trace.

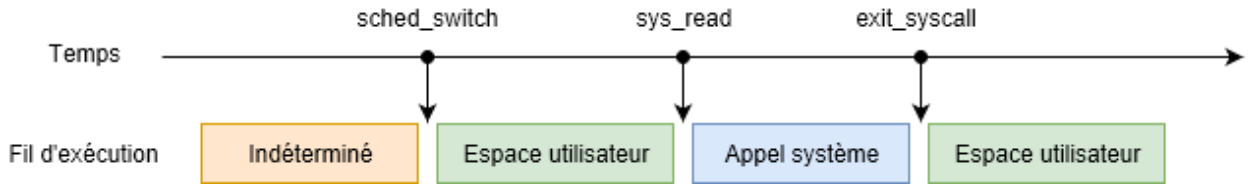

Figure 2.1 Exemple de conversion d'évènements en états

Un problème lié à cette méthode est lorsqu'il y a eu des pertes d'évènements lors de la session de traçage. Dans un tel cas, un état peut être incohérent durant une durée indéterminée. Dans la figure 2.1, si l'évènement `sys_read` était perdu, le fil d'exécution serait resté dans l'état `Espace utilisateur` et il n'y aurait pas eu d'état `Appel système`. Afin d'initialiser l'état des fils d'exécution, les traceurs ETW et LTTng propagent une série d'évènements, détaillant l'état courant du système, au début et à la fin de la trace. Ils peuvent également indiquer le nombre d'évènements perdus afin de valider la fiabilité des informations de la trace.

2.2.2 Sauvegarder l'information d'état dans une structure de données

Pour une trace de grande taille, le coût associé à la construction de la machine à état est considérable. Chaque requête vers le gestionnaire d'état, pour connaître l'état d'un fil à un instant t, implique de devoir relire la trace depuis le début, ce qui s'avère être inefficace. Pour remédier à ce problème, Montplaisir et al. proposent de sauvegarder les intervalles d'états, pour toute la durée de temps couverte par la trace, dans une structure en arbre appelée *State History Tree* (Montplaisir et al., 2013). Leur solution est fondée sur l'hypothèse que

les intervalles d'états arrivent en ordre croissant de temps de fin. Les critères de recherche selon le temps sont garantis et l'arbre n'a pas à être rééquilibré pendant sa construction. L'écriture de l'arbre sur disque se fait de manière incrémentale et les recherches dans l'arbre s'exécutent en temps logarithmique. Cette solution supporte des traces de l'ordre de dizaines de gigaoctets.

Une autre méthode, proposée par Chan et al., serait de sauvegarder les états dans la trace elle-même (Chan et al., 2008). Ainsi, la trace contient à la fois des évènements ponctuels et des états. Cependant, leur approche n'est pas très flexible, puisque les informations d'états du traceur sont fortement couplées à l'affichage.

2.2.3 Outils de visualisation de trace

Il existe une multitude d'outils pour la visualisation de traces. Certains se spécialisent pour un certain format de trace ou un certain système, alors que d'autres offrent plus de flexibilité et d'extensibilité. Quelques outils seront présentés dans cette section.

`LTTng-Scope` [2] est une application multiplateforme basée sur JavaFX pour la visualisation de traces au format CTF. Le découplage entre la visualisation et l'analyse de trace est bien défini. L'application utilise une librairie implémentée en Java, appelée *Jabberwocky* [3], pour l'analyse de grandes traces. La librairie propose quelques analyses dont l'état des fils d'exécution, l'utilisation des ressources de l'unité centrale de traitement (*Central Processing Unit (CPU)*) et le décompte des évènements à travers le temps. *Jabberwocky* est générique et extensible, mais ne supporte que le format CTF pour le moment. De plus, les fonctionnalités de LTTng-Scope sont encore très limitées puisque l'outil est encore en version bêta.

`Trace Compass` est un outil multiplateforme de visualisation de traces basé sur Eclipse (Prieur-Drevon et al., 2018). L'outil permet d'analyser les traces au format GNU Debugger (GDB), Best Trace Format (BTF), CTF et Packet CAPture (PCAP). Plusieurs analyses, telles que l'état des fils d'exécution, les utilisations des ressources CPU, l'utilisation de la mémoire, l'activité des disques, les compteurs de performance, ou encore le chemin critique, sont offertes. Les travaux de Kouamé ont permis aux utilisateurs de définir leurs propres analyses par le biais d'un fichier XML qui doit suivre une certaine structure (Kouamé, 2015). L'outil supporte de grandes traces, mais son architecture est monolithique et plusieurs composantes de l'interface utilisateur sont fortement couplées aux composantes responsables d'analyser les traces.

Google a développé sa propre infrastructure de traçage dans Chromium, permettant ainsi

2. https ://github.com/lttng/lttng-scope
3. https ://github.com/efficios/jabberwocky

aux développeurs d'identifier des problèmes de performance de leur application web (Prieur-Drevon et al., 2018). Le traceur de Chromium produit des traces au format Trace Event (Duca and Sinclair), et l'infrastructure propose des analyses telles que l'état des fils d'exécution, la pile d'appels, le chemin critique pour le rendu de page HTML, l'utilisation de la mémoire ou encore l'utilisation des ressources CPU. `Trace-Viewer` est l'outil de visualisation de trace intégré à Chromium, écrit en JavaScript, et accessible à l'adresse `chrome://tracing`. Le format Trace Event est nativement supporté et le support pour d'autres formats exige d'écrire un *trace importer*. Cependant, la trace est complètement chargée en mémoire, ce qui pose un problème de mise à l'échelle lorsque la taille de la trace dépasse un certain seuil.

`OpenZipkin` est un système de traçage distribué développé par Twitter (Schulz et al., 2008). La composante responsable du traçage est une implémentation de OpenTracing écrite en Java. La composante de visualisation, écrite en JavaScript, est accessible par le biais d'un navigateur web. L'architecture est organisée selon un modèle client-serveur (voir section 2.4.1) où le client communique avec le serveur par le biais de requêtes HTTP à un API REST. Malheureusement, cet outil se limite à OpenTracing pour le moment et ne supporte pas d'autres formats de trace. De plus, la gestion des traces qui contiennent plusieurs millions de *spans* n'est pas encore supportée.

`TraceAlyzer` est un outil de visualisation de trace propriétaire destiné à être utilisé pour les systèmes temps réel ou les systèmes embarqués basés sous Linux (Kraft et al., 2010). L'outil supporte entre autres les systèmes d'exploitation FreeRTOS, Wittenstein SafeRTOS, Wind River VxWorks et Micrium μC/OS. TraceAlyzer offre plusieurs vues permettant de comprendre l'ordonnancement des tâches, les changements de priorités ou encore le verrouillage de sections critiques. Néanmoins, les traces sont stockées en mémoire, ce qui limite l'outil pour l'analyse et la visualisation de grandes traces (Prieur-Drevon, 2017).

`Jumpshot` est l'outil de visualisation de traces de la suite MPE écrit en Java (Zaki et al., 1999). L'outil montre l'état de l'ensemble des fils d'exécutions distribués, les messages échangés entre ces derniers, et le temps passé en calcul comparativement au temps passé en communication. Les messages sont représentés par des flèches, partant du fil émetteur au fil récepteur. Malheureusement, les vues de Jumpshot sont étroitement liées à la structure de données SLOG-2, qui est également un format de trace.

2.3 Récents travaux reliés aux EDI

2.3.1 Nouveaux EDI

`Visual Studio Code`[4] (VS Code) est l'éditeur de code source, libre et multiplateforme, développé par Microsoft. L'outil offre plusieurs fonctionnalités telles que le débogage, la gestion de versions du code sous Git, l'auto complétion et la réingénierie du code (*refactoring*). Il intègre également une console et les utilisateurs peuvent facilement personnaliser l'outil en installant des extensions, et en changeant le thème et les raccourcis. VS Code est écrit en TypeScript, un langage typé qui est ensuite transcompilé en JavaScript. L'EDI se base sur Electron, un cadriciel (*framework*) permettant de développer des applications natives multiplateformes avec des technologies web (HTML, CSS, JavaScript). D'ailleurs, l'éditeur de code Atom repose également sur Electron. Finalement, VS Code implémente le *Language Server Protocol* (voir section 2.3.2).

Efftinge and Kosyakov ont annoncé le projet `Theia`[5] lors de l'EclipseCon qui s'est tenu en France en 2017 (Efftinge and Kosyakov, 2017). Theia est à la fois un cadriciel libre pour le développement d'EDI et un EDI multiplateforme. Le projet partage beaucoup de similarité avec VS Code. En effet, Theia est écrit en TypeScript, utilise Electron, Node.js et supporte le *Language Server Protocol*. D'ailleurs, plusieurs composantes de Theia telles que l'éditeur Monaco et le terminal `xterm.js` proviennent de VS Code. Cependant, contrairement à VS Code, Theia supporte un déploiement hybride. En plus de pouvoir être déployé comme une application Electron, il est possible de déployer Theia dans l'infonuagique et avoir l'interface graphique dans un navigateur web.

2.3.2 Fonctionnalités de langages

Le `Language Server Protocol` (LSP) est un protocole de communication ouvert basé sur le JSON-RPC (voir section 2.5.2), développé par Microsoft, entre un éditeur ou un EDI et un serveur de langage de programmation (Keidel et al., 2016). Un serveur de langage offre des fonctionnalités telles que l'autocomplétion de code, la coloration syntaxique (*syntax highlighting*), l'atteinte vers la définition (*go-to definition*) ou encore l'affichage des erreurs du code. L'idée du LSP est de standardiser la façon dont les serveurs de langage mettent ces fonctionnalités à la disposition des EDI. Ce faisant, l'implémentation d'un EDI est complètement découplée d'un serveur de langage et le LSP favorise la réutilisation des serveurs de langage dans d'autres EDI. Le protocole a été initialement développé pour VS Code, mais

4. https ://github.com/Microsoft/vscode
5. https ://github.com/theia-ide/theia

d'autres organisations telles que Red Hat et Codeenvy ont contribué à la spécification du LSP. D'ailleurs, plusieurs organisations telles que la fondation Eclipse, Github, Sourcegraph et Red Hat ont adapté leurs outils de développement populaires (Eclipse, Atom, IntelliJ, Emacs, Sublime, etc.) pour supporter le protocole. Au moment de la rédaction de ce mémoire, il existe 46 langages de programmation (dont le Java, C/C++, C#, PHP, Python et JavaScript) qui offrent un support pour le LSP.

Trois types de messages sont définis par le LSP : une requête, une réponse et une notification. Une requête est faite du client vers le serveur par le biais d'un appel de procédure JSON-RPC. Le serveur répond à la requête par une réponse ; une requête doit toujours être répondue par une réponse. Une notification peut être envoyée depuis le serveur ou le client et une réponse n'est pas obligatoire. La figure 2.2 illustre un exemple d'interaction et d'échanges de messages entre un EDI et un serveur de langage durant une séance d'édition de code.

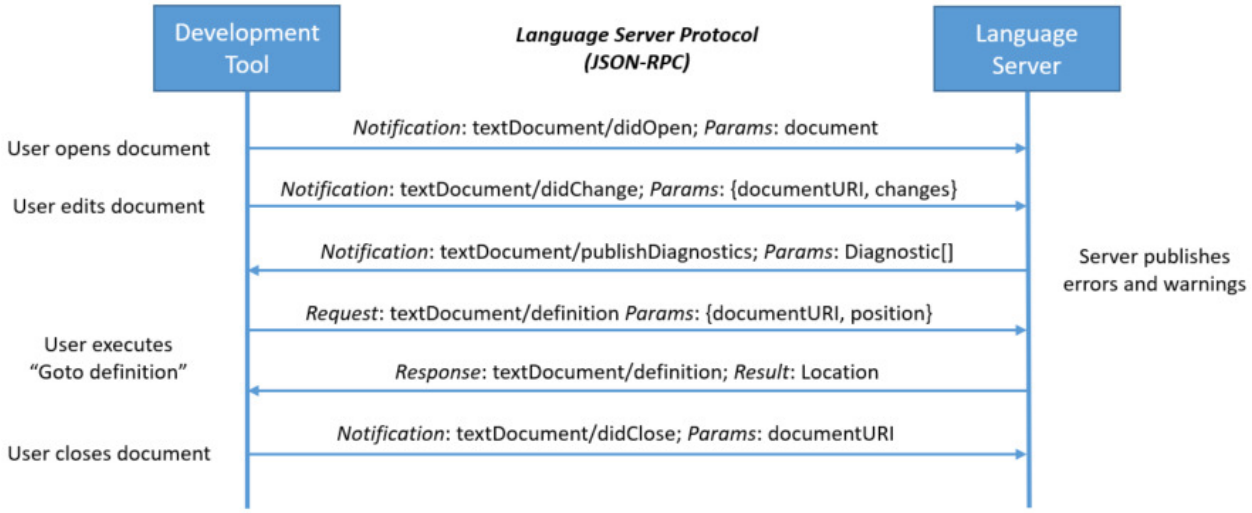

Figure 2.2 Exemple d'interaction entre un EDI et un serveur de langage

Source : https ://microsoft.github.io/language-server-protocol/overview

Les interactions entre le client et le serveur utilisent des modèles neutres, c'est-à-dire qui ne sont ni dépendants du langage de programmation ni de l'EDI. Par exemple, lorsque le client est intéressé à atteindre la définition d'une procédure, le client et le serveur s'échangent un *Uniform Resource Identifier* (URI) permettant de localiser un document sur disque et une position dans le document. D'ailleurs, le LSP fournit un mécanisme, lorsqu'un serveur de langage ne peut supporter toutes les fonctionnalités définies par le protocole. Le serveur doit initialement exposer au client ce qu'il supporte et ce qu'il ne supporte pas. À l'heure actuelle, le protocole indique qu'il s'agit de la responsabilité du client de gérer et maintenir la durée

de vie du serveur. Finalement, le LSP ne spécifie pas comment les messages sont transférés du client au serveur. Ces deux composantes peuvent être issues du même processus ou de multiples processus distribués communiquant à travers un *socket* réseau.

`Monto` est une architecture mettant en relation un EDI et des composantes qui fournissent des fonctionnalités de langage (Keidel et al., 2016). Les auteurs ont choisi une approche d'architecture orientée service (voir section 2.4.1) où chaque fonctionnalité de langage telle que l'autocomplétion est offerte par un service. Chaque service peut être à son tour décomposé en sous-services. L'architecture prévoit, comme le LSP, un mécanisme pour que l'EDI connaisse les fonctionnalités disponibles : les services doivent préalablement s'enregistrer auprès d'une composante appelée *broker*. Un module d'extension au sein de l'EDI est responsable de communiquer avec le *broker*. La figure 2.3 illustre une vue de haut niveau d'interaction entre les composantes de l'architecture.

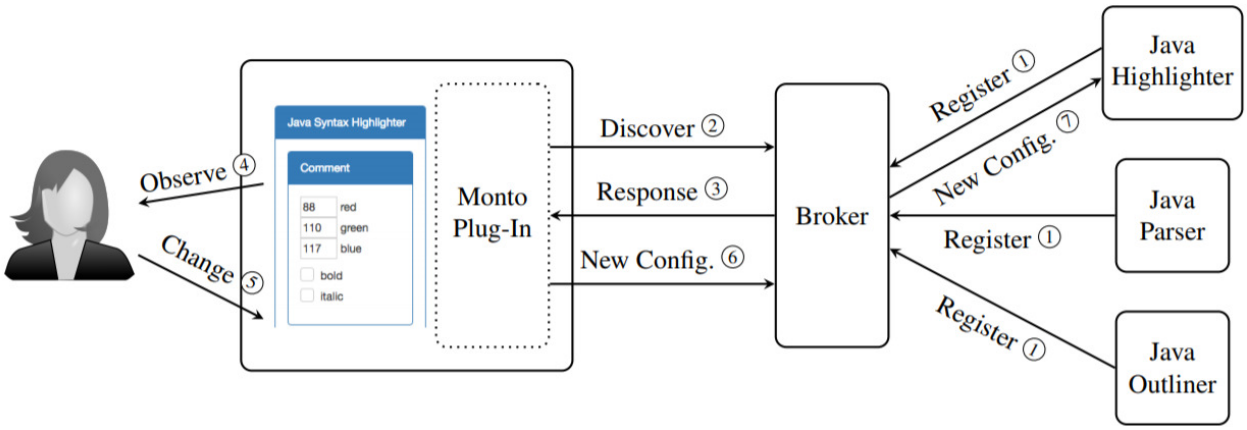

Figure 2.3 Interaction entre les composantes de l'architecture Monto (Keidel et al., 2016)

L'architecture définit deux types de messages : *source message* et *product message*. Le *source message*, envoyé depuis le module d'extension de l'EDI vers le *broker*, contient un identifiant unique, le nom du fichier, le langage de programmation et le contenu du fichier. Le *broker* est responsable de distribuer les *source messages* aux services pertinents et de répondre au module d'extension de l'EDI lorsque les opérations sont terminées. Le *product message* encapsule une représentation intermédiaire générique, un modèle indépendant du langage et de l'EDI. Les communications se font par le biais de *ZeroMQ*, une bibliothèque de messagerie asynchrone haute performance, et tous les messages sont encodés sous le format JavaScript Object Notation (JSON). L'approche proposée par les auteurs permet d'avoir un serveur sans état (*stateless*). De plus, Monto n'a pas besoin de gérer et maintenir une copie des documents ouverts, comme c'est le cas pour le LSP. Cependant, envoyer tout le contenu d'un

fichier au format JSON n'est pas nécessairement optimal en termes de données à transférer sur le réseau.

2.3.3 Fonctionnalités de débogage

VS Code utilise un mécanisme appelé `Debug Adapter Protocol` (DAP) pour le support des fonctionnalités de débogage. Puisque l'interface graphique de débogage n'est pas spécifique à un langage de programmation, l'EDI n'interagit pas directement avec le débogueur, mais avec un module d'extension appelé `debug extension`. La figure 2.4 illustre les composantes impliquées pour le support du débogage dans VS Code.

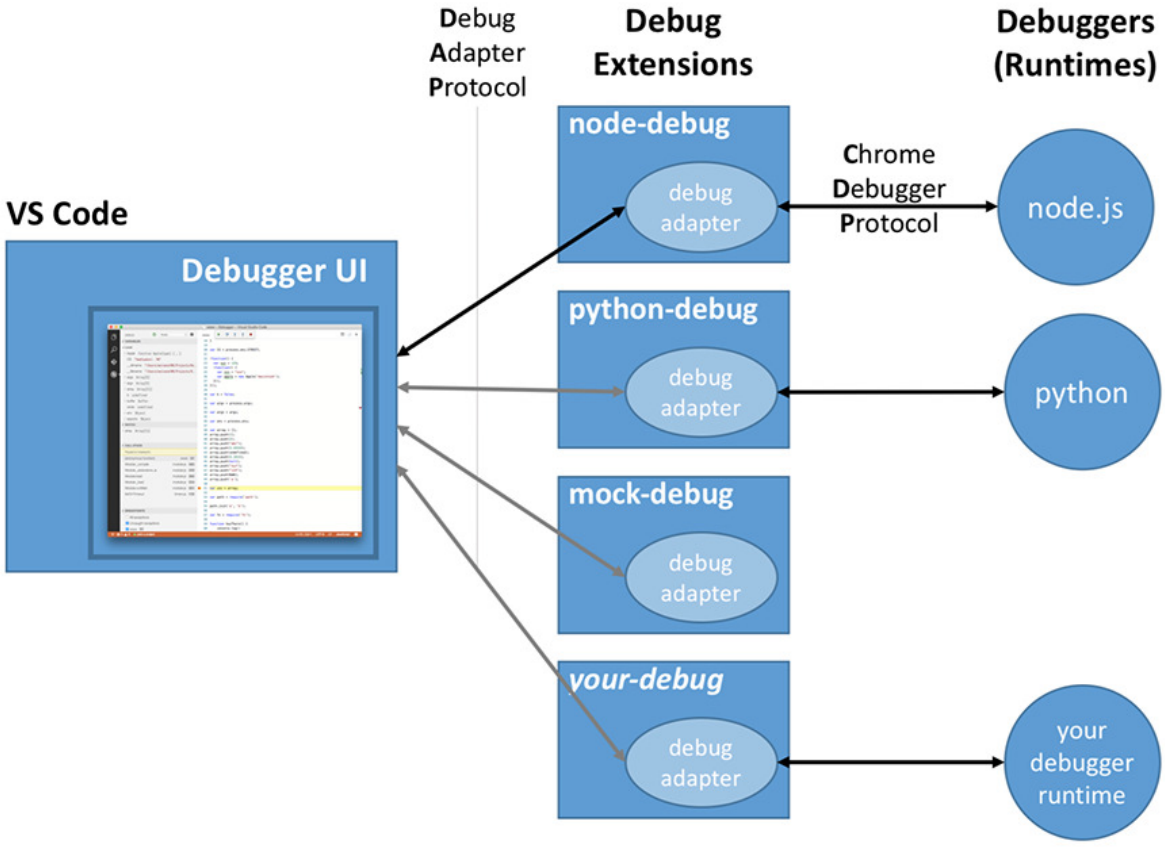

Figure 2.4 Mécanisme de support pour le débogage dans VS Code

Source : https ://code.visualstudio.com/docs/extensions/example-debuggers

Le `debug extension` comprend le `debug adapter`, la composante responsable d'appeler directement l'API du débogueur en fonction d'un appel du DAP. Par exemple, lorsque l'utilisa-

teur active un point d'arrêt, VS Code envoie une requête de type `setBreakPoint` par le biais du DAP et le *debug adapter* saura quelle procédure du débogueur appeler. Dans le cas de Node.js, le `debug adapter` traduit simplement le DAP et communique avec le débogueur par le biais du `Chrome Debugger Protocol`. Toute communication entre VS Code et le module d'extension se fait par le biais des entrées-sorties standard (*stdin, stdout*) au format JSON. Le `debug adapter` est généralement un exécutable autonome écrit dans le même langage de programmation que le débogueur.

`Kómpos` est un protocole de communication bidirectionnel entre un EDI et un débogueur qui supporte de multiples modèles de concurrence (Marr et al., 2017). Par exemple, dans les langages tels que le Java, le C++ et le C#, il existe un modèle de concurrence pour définir l'exécution simultanée et le contrôle des ressources partagées. Ce modèle comprend les verrous et les fils d'exécution (*Threads and Locks* (T&L)). Dans d'autres langages, tels que le JavaScript, il existe un autre modèle, appelé *Communicating Event Loop* (CEL), pour la concurrence qui se base sur la boucle d'évènements (*event loop*), les promesses et les messages asynchrones. Les auteurs ont également présenté les modèles *Communicating Sequential Processes* (CSP), *Software Transactional Memory* (STM) et *Fork/Join Parallelism* (F/J). La figure 2.5 liste les cinq modèles de concurrence et leurs concepts.

	T&L	CEL	CSP	STM	F/J
Activities	threads	actors	processes	threads	tasks
Dynamic Scopes	object monitors	turns		transactions	
Passive Entities	conditions locks	messages promises	channels messages		
Send Operations	acquire lock signal condition	send message resolve promise	send message		
Receive Operations	release lock wait for condition join thread		receive message join process	join thread	join task

Figure 2.5 Concepts de chaque modèle de concurrence (Marr et al., 2017)

Les auteurs ont extrait des modèles génériques, listés à gauche de la figure, pour abstraire à L'EDI les différents concepts des cinq modèles de concurrence. De plus, les modèles ne sont pas spécifiques à un langage de programmation ni à l'interface graphique de l'EDI. Le protocole définit quatre types de messages : `Source`, `BreakpointUpdate`, `Stopped` et `Step`. Le premier sert à échanger l'URI de l'emplacement du fichier à déboguer. Le deuxième est

pour définir l'emplacement d'un point d'arrêt. Le troisième sert à notifier l'EDI lorsqu'un point d'arrêt est atteint ou qu'une opération pas-à-pas est terminée. Finalement, le dernier est pour indiquer qu'il faut reprendre l'exécution. En plus des modèles génériques qui sont échangés entre l'EDI et le débogueur, des métadonnées sont transmises afin d'avoir plus de détails lors de la session de débogage. `Kómpos` ne spécifie pas de couche de transport, mais requiert que la communication soit bidirectionnelle. Les auteurs ont choisi de l'implémenter avec les Web Sockets et le format d'échange est le JSON.

2.4 Architecture logicielle

2.4.1 Styles architecturaux

Il existe plusieurs façons d'organiser les composantes et les connecteurs. Certaines configurations, qui partagent des similarités, peuvent être associées à un même modèle que l'on appelle style architectural. Ainsi, un style architectural décrit un patron de haut niveau pour la structure et l'organisation des composantes et des connecteurs. En pratique, les systèmes n'utilisent pas exclusivement un seul style architectural, mais en combinent plusieurs. Cette section décrit les styles architecturaux les plus communs.

Architecture multicouche

L'architecture multicouche (Garlan and Shaw, 1993) organise hiérarchiquement les composantes d'un système en couches. Dans cette approche, une couche logicielle de niveau `N` n'interagit qu'avec les couches de niveau `N-1` et `N+1`. L'interaction entre chaque couche est définie soit par un protocole de communication ou soit par une invocation explicite de procédures définies dans une interface. Ce style architectural convient adéquatement aux systèmes qui admettent plusieurs niveaux d'abstraction. Ainsi, ces systèmes peuvent diviser un problème complexe en une série d'étapes et d'opérations plus courtes. Un autre avantage de cette architecture est la réutilisabilité. Chaque couche peut admettre différentes implémentations interchangeables. Cependant, en pratique, ce ne sont pas tous les systèmes qui peuvent être facilement représentés et structurés en couches.

Concrètement, le modèle OSI (*Open Systems Interconnection*) en réseau est un exemple connu d'application de l'architecture multicouche.

Architecture orientée objet

L'architecture orientée objet (Garlan and Shaw, 1993) représente les données et les opérations qui s'y rattachent en structures de données. Dans cette architecture, toutes composantes identifiables d'un système telles que les opérations d'entrées-sorties, les fichiers ou les fils d'exécutions sont représentés par des structures de données en mémoire appelées objets et qui offrent une description de haut niveau aux utilisateurs. Certains langages de programmation tels que le Java sont fondés sur le paradigme orienté objet, ce qui rend l'architecture orientée objet implicite. En revanche, d'autres langages tels que les langages procéduraux ou fonctionnels n'utilisent pas les notions d'objets.

Architecture centrée sur les données

L'architecture centrée sur les données (Tanenbaum and Steen, 2006) est fondée sur l'idée que les composantes logicielles se synchronisent et communiquent à travers un référentiel commun. Ce référentiel peut être une structure de données centrale, une base de données ou encore un système de fichiers distribués. Les composantes sont indépendantes et peuvent ajouter, supprimer ou modifier les données du référentiel. Dans ce style architectural, le changement d'implémentation d'une composante A n'affecte pas le fonctionnement d'une composante B. De plus, les composantes n'ont pas à se connaître, mais sont fortement dépendantes aux changements du référentiel.

Architecture basée sur les évènements

L'architecture basée sur les évènements (Garlan and Shaw, 1993) repose sur le principe d'invocation implicite entre les composantes logicielles. Classiquement, dans l'approche d'invocation explicite, une composante A interagit avec une composante B en appelant explicitement une procédure exposée par B. Dans l'approche d'invocation implicite, une composante A interagit avec une composante B en diffusant un ou plusieurs évènements. Selon l'intérêt de la composante B pour les évènements produits par A, le système exécutera certaines routines de B. Toute autre composante intéressée par les évènements produits par A n'a qu'à s'enregistrer auprès du système et à fournir une routine à exécuter lorsque le système reçoit les évènements.

Dans cette approche, les composantes logicielles qui diffusent les évènements ne connaissent pas les composantes intéressées, ni les routines exécutées, ni leur statut de progression. De plus, l'ordre dans lequel les routines des composantes intéressées s'exécutent n'est pas nécessairement déterministe. Ces éléments constituent les principaux désavantages de l'approche

implicite. Cependant, cette approche facilite l'évolution du système et les composantes logicielles sont fortement réutilisables. Finalement, l'architecture basée sur les évènements n'utilise pas exclusivement l'approche d'invocation implicite, mais plutôt un mélange des deux approches présentées.

Concrètement, les interfaces utilisateurs, les systèmes de gestion de base de données et les environnements de développement intégrés sont des exemples de systèmes qui utilisent le style d'architecture basée sur les évènements.

Architecture client-serveur

L'architecture client-serveur (Coulouris et al., 2011) est historiquement la plus importante et la plus utilisée pour les systèmes distribués. Dans ce style architectural, deux entités se distinguent par leurs responsabilités : un client qui veut accéder à une ressource et un serveur qui gère les ressources. Généralement, un serveur est une machine ou un processus doté d'une bonne puissance de calcul et qui peut répondre à plusieurs requêtes concurrentes de clients. Les clients n'interagissent pas directement entre eux, mais par le biais du serveur. D'ailleurs, les clients ne sont pas nécessairement sur la même machine physique que le serveur, ce qui implique un protocole de communication entre le client et le serveur. Finalement, un serveur peut être à son tour un client d'un autre serveur. Par exemple, un serveur web est un serveur pour un navigateur web, mais devient client pour une base de données.

L'architecture client-serveur se décline en plusieurs sous-catégories telles que le client lourd et le client léger. Dans la première, le client a amplement plus de responsabilités en ce qui concerne les calculs et le traitement des données, ce qui réduit la responsabilité du serveur à simplement envoyer l'information demandée. Dans la deuxième, c'est l'inverse, le serveur est responsable des calculs et du traitement des données, ce qui laisse seulement la responsabilité d'affichage au client. Ces deux sous-catégories visent à mitiger les conséquences reliées à la panne du serveur.

Architecture pair-à-pair

L'architecture pair-à-pair (Coulouris et al., 2011) ne fait pas de distinction de rôle et responsabilité entre ses entités que l'on appelle nœuds. Dans ce style architectural, les nœuds sont répartis dans un réseau décentralisé où chacun joue le même rôle dans l'exécution d'une tâche. Les nœuds travaillent de façon coopérative, mais n'ont pas nécessairement la même puissance de calcul. De plus, ils n'ont pas nécessairement les mêmes ressources, ce qui implique qu'un nœud peut à la fois demander et fournir des ressources. Contrairement à l'architecture client-

serveur, l'architecture pair-à-pair qui regroupe plusieurs nœuds est plus résiliente lorsqu'une panne d'un nœud survient.

Les systèmes de calcul à haute performance, les systèmes de partage de fichiers et la chaîne de blocs (*blockchain*) sont des exemples d'applications qui exploitent l'architecture pair-à-pair.

Architecture orientée services

L'architecture orientée services (Coulouris et al., 2011) repose sur une collection de services simples, indépendants et réutilisables qui interagissent entre eux. Un service est une fonctionnalité ou une fonction qui produit des résultats. Dans ce style architectural, il existe deux rôles : le fournisseur de service et le consommateur de service. Le consommateur communique avec le fournisseur par le biais d'un standard, ce qui assure le découplage et l'interchangeabilité entre ces deux entités. Le consommateur n'est intéressé que par le résultat final, ce qui abstrait totalement l'implémentation du fournisseur. De plus, le standard de communication qui relie le consommateur et le fournisseur assure l'interopérabilité entre ces derniers.

2.4.2 Patron de conception

Un patron de conception décrit comment organiser des composantes entre elles afin de résoudre un problème de conception logicielle (Gamma et al., 1995). Formalisés en 1995 dans l'ouvrage *Design Patterns – Elements of Reusable Object-Oriented Software*, les patrons de conception se présentent comme suit :

— Un nom ;
— Une description du problème que le patron résoud ;
— Une description de la solution ;
— Les conséquences du patron ;

Les patrons du *Gang of Four* (GoF) regroupent 23 patrons de conception pour les systèmes orientés objet. Les patrons sont classés en trois familles : créateurs, structuraux et comportementaux (Gamma et al., 1995).

Les patrons GRASP (*General responsibility assignment software patterns*) regroupent neuf directives dans l'attribution des responsabilités aux classes dans un système orienté objet (Larman, 2012). Les règles présentées sont : expert, créateur, faible couplage, forte cohésion, contrôleur, polymorphisme, indirection, fabrication pure et protection des variations.

Dans leur ouvrage *Cloud Design Patterns*, Homer et al. présentent 32 patrons de conception pour les systèmes distribués (Homer et al., 2014). Ces patrons visent à répondre aux défis

liés aux systèmes répartis : disponibilité, gestion des données, performance et mise à l'échelle, résilience, communication et messagerie, sécurité, gestion et surveillance.

2.5 Interopérabilité des systèmes hétérogènes

L'interopérabilité est la capacité d'un système à échanger des données avec un autre sans qu'ils soient écrits avec le même langage de programmation. Pour ce faire, les systèmes peuvent interagir indirectement en utilisant un référentiel commun ou directement par le biais d'un protocole de communication réseau. Dans le premier cas, les systèmes se synchronisent à travers une base de données, un système de fichiers distribués ou encore un espace de mémoire partagé. Dans le second cas, les systèmes utilisent des protocoles tels que TCP, UDP ou HTTP et échangent les données sous un format commun. Cette section ne s'intéresse qu'aux techniques d'interaction directe entre les systèmes.

2.5.1 Sérialisation

La sérialisation est un processus d'écriture par lequel l'état d'un objet est transformé en flux (Hericko et al., 2003). La désérialisation est le processus inverse qui rebâtit un objet à partir du flux. Ce flux est soit textuel ou soit binaire, et plusieurs formats de sérialisation existent pour organiser et structurer les données. Nous présentons ici quelques formats de sérialisation supportés par la plupart des langages de programmation populaires.

Format textuel

Le `Extensible Markup Language` (XML), dérivé du Standard General Markup Language (SGML), organise ses données selon des balises (Bray et al., 1997) appelées noeuds. La structure d'un document XML est définie par un schéma et contient plusieurs balises pouvant être organisées comme une structure arborescente. Un noeud peut contenir des attributs. Ci-dessous, un exemple de document XML.

Code 2.1 Exemple de document XML

```
<?xml version="1.0" encoding="UTF-8"?>
<trace>
  <event start="0" end="1">sched_switch</event>
  <event start="1" end="2">sys_read</event>
  <event start="2" end="3">exit_syscall</event>
</trace>
```

Le `JavaScript Object Notation` (JSON) est un format de données fortement inspiré de la notation des objets du JavaScript (International, 2017). Ce format repose sur deux éléments structuels : des ensembles de paires (clé-valeur) et des listes de valeurs ordonnées. Ci-dessous, un exemple de document au format JSON.

Code 2.2 Exemple de document JSON

```
{
 trace: {
     events: [
          { start: "0", end="1", name="sched_switch" },
          { start: "1", end="2", name="sys_read" },
          { start: "2", end="3", name="exit_syscall" }
        ]
    }
}
```

Format binaire

Le `Binary JSON` (BSON) est une représentation binaire du JSON et utilisée au sein de MongoDB pour le transfert et le stockage des données (MongoDB, 2018). Ce format permet de représenter des structures de données simples et inclut des types additionnels tels que `int`, `long`, `date` et `floating point`.

Le format de sérialisation de Google, les `Protocol Buffers` (Protobuf) sont un format binaire avec un langage de description d'interface (*Interface Definition Language* (IDL)) (Google, 2018). L'utilisateur définit dans un fichier `.proto` les structures de données (appelées messages) à échanger et le type de chaque propriété de la structure (entier, chaîne de caractères, etc.). Ensuite, le compilateur `protoc` génère à partir de ce fichier le code source des classes ou composantes logicielles qui envoient et reçoivent ces structures de données. Protobuf utilise les `varints` pour l'encodage des entiers. Cette technique consiste à utiliser un nombre variable d'octets, supprimant les octets plus significatifs qui sont nuls, pour représenter un nombre.

`Apache Thrift` est un cadriciel RPC avec un IDL développé par Facebook qui offre un format de sérialisation binaire (Slee et al., 2007). L'utilisateur définit dans un fichier `.thrift` les structures de données à échanger et le compilateur Thrift génère le code source des classes. Thrift offre deux formats de sérialisation : `TBinaryProtocol` et `TCompactProtocol` (Prunicki, 2009). Ce dernier produit des flux plus compacts et utilise diverses optimisations telles que

les `varints` pour l'encodage des entiers.

2.5.2 Remote Procedure Call et Remote Method Invocation

Les Remote Procedure Call (RPC) permettent à une machine A d'appeler des procédures sur une machine distante B, comme si elles étaient disponibles dans l'espace d'adressage local (Coulouris et al., 2011). Les Remote Method Invocation (RMI) suivent le même principe que les RPC, mais s'appliquent aux objets.

`Common Object Request Broker Architecture` (CORBA) est une architecture définie par l'*Object Management Group* pour standardiser la façon dont les systèmes hétérogènes communiquent (Siegel, 1998). CORBA utilise des modèles orientés objet et l'utilisateur doit en définir les interfaces dans un IDL. Ces interfaces produisent deux composantes qui jouent le rôle de proxy : le *stub* et le *skeleton*. La communication entre ces deux composantes est gérée par le *Object Request Broker* (ORB). L'ORB est responsable de gérer tous les détails concernant le routage d'une requête du client vers le serveur, et le routage de la réponse vers sa destination.

`JSON-RPC` est un protocole RPC sans état, qui utilise le format JSON comme format d'échange et est conçu pour être simple (Group et al., 2013). Une requête du client contient les champs `jsonrpc` pour spécifier la version du protocole, `method` pour le nom de la méthode à exécuter, `params` pour lister les valeurs des paramètres de la méthode et `id` pour associer une réponse du serveur à la requête du client. Une requête doit toujours recevoir une réponse. Le protocole supporte également les notifications, un message qui n'a pas besoin d'avoir de réponse. Finalement, JSON-RPC ne spécifie pas la couche de transport par laquelle les messages sont acheminés.

`gRPC` est un cadriciel RPC haute performance développé par Google qui utilise le format d'échange Protobuf (gRPC authors). L'utilisateur définit des services dans un fichier `.proto` et le compilateur `protoc` génère les API qui seront utilisés par le client et le serveur. gRPC utilise le protocole HTTP/2 comme couche de transport et offre des fonctionnalités pour la gestion de l'authentification, des erreurs et du débogage.

`Java-RMI` est un API qui permet à une application Java d'exécuter une méthode d'objet qui peut se trouver sur une machine distante (Waldo, 1998). Le système appelant et le système appelé doivent nécessairement être écrits en Java. Toutes les fonctionnalités supportées, telles que la sérialisation des classes Java et le ramasse-miettes distribué, se retrouvent dans le paquet `java.rmi`.

2.5.3 Services web

Un service web est un logiciel conçu spécialement pour supporter les interactions entre des machines sur un réseau (Group et al., 2004). L'interface d'un service est décrite selon un certain format tel que le *Web Services Description Language* (WSDL). Les communications entre ces machines se font par le protocole HTTP et utilisent soit le format XML ou JSON.

`Single Object Access Protocol` (SOAP) est un protocole RPC qui utilise le XML comme format d'échange (Box et al., 2000). Les messages échangés pour l'invocation des méthodes d'objets doivent suivre un ensemble de règles et une certaine structure. En pratique, le transfert de ces messages se fait par le biais du protocole HTTP, mais d'autres protocoles comme le SMTP peuvent être utilisés. SOAP est constitué de trois parties :

1. Une enveloppe qui définit la structure du message et comment le traiter ;
2. Un en-tête qui contient des informations supplémentaires facultatives ;
3. Un corps qui contient les informations sur l'appel de méthode, et la réponse ;

`Representational State Transfer` (REST) est un style architectural qui définit six contraintes pour les systèmes hypermédia distribués (Fielding, 2000). En respectant ces contraintes, les services gagnent en performance, simplicité, portabilité et mise à l'échelle. Les six contraintes sont : 1) Client-serveur ; 2) Sans état ; 3) Avec mise en cache ; 4) En couches ; 5) Avec code à la demande ; 6) À interface uniforme ;

Dans ce style architectural, une *ressource* est l'abstraction de toute forme d'information qui a un nom (document, image) et un client peut la manipuler par le biais d'un *Uniform Resource Locator* (URL). L'état d'une ressource à un temps donné, qui est appelé représentation, s'échange selon un certain format (JSON, JPEG, etc.). En pratique, REST est utilisé avec le protocole HTTP, mais le style architectural ne définit aucune directive d'implémentation ni de protocole.

2.6 Conclusion de la revue de littérature

La présentation des récents travaux sur les EDI nous a permis de constater que ces outils tendent à utiliser une approche client-serveur où les fonctionnalités majeures de langage et de débogage se retrouvent du côté serveur. L'interface graphique de l'EDI utilise des modèles neutres et n'a donc pas de couplage avec un langage de programmation en particulier. La communication entre le client et le serveur se fait par le biais d'un protocole de communications et le format de sérialisation choisi est le JSON. Cette approche favorise la réutilisation des serveurs dans d'autres EDI, tout en faisant abstraction de leur fonctionnement interne.

CHAPITRE 3 MÉTHODOLOGIE

Cette section décrit les démarches suivies afin de répondre à la question de recherche tout en satisfaisant les objectifs de recherche. Les étapes de notre démarche sont itératives et incrémentales.

3.1 Identification des besoins

La revue de littérature a présenté différents styles architecturaux et a montré que les récents travaux reliés aux EDI semblent se tourner vers une approche client-serveur où un protocole de communication standard a été introduit. Afin d'appliquer la même approche à notre problème, la première étape consiste à déterminer les responsabilités du client et du serveur. Cette séparation déterminera par la suite quelles classes ou composantes seront nécessaires aux deux entités. Ensuite, la deuxième étape vise à comprendre comment Trace Compass a été conçu, comment les classes, les composantes et les modules interagissent entre eux. Cette étape est primordiale avant de proposer une nouvelle architecture. Nous avons choisi Trace Compass puisque l'outil est *open source*, supporte plusieurs formats de traces et supporte l'analyse de traces de l'ordre du gigaoctet. Pour comprendre l'architecture de Trace Compass, nous devons utiliser des outils de rétro-ingénierie (*reverse engineering*) du code, consulter le code source et effectuer un transfert de connaissance auprès de l'équipe de développement. Après, la troisième étape cherche à déterminer comment le client et le serveur communiqueront. Le protocole introduit doit admettre un faible surcoût d'exécution et doit minimiser la quantité de données à transférer. La section de la revue de littérature portant sur l'interopérabilité des systèmes nous sera fortement utile. Finalement, la dernière étape de cette phase consiste à déterminer l'architecture du client, ses classes et ses composantes.

À la fin de la phase d'identification des besoins, nous proposons une version du plan de la nouvelle architecture qui inclut, entre autres, des diagrammes de classes et la spécification du protocole.

3.2 Implémentation et réingénierie

Avant de procéder à toute modification du code source de Trace Compass, nous devons écrire des tests unitaires afin de prévenir toute régression. Dans un premier temps, il faut déterminer quelles seront les méthodes, classes et fonctionnalités à tester. Dans un deuxième temps, une fois les tests unitaires écrits, nous procédons à l'implémentation du plan de la nouvelle

architecture et à la réingénierie de Trace Compass. Les tests unitaires permettront également de vérifier et valider que la réingénierie n'introduit pas de nouveaux bogues. Puisque les récents travaux sur les EDI se tournent vers des technologies web, nous choisissons Trace Compass comme étant le serveur ; le client sera une application web écrite en HTML, CSS et JavaScript s'exécutant sur un navigateur web. Dans un troisième temps, nous devons déterminer quelles seront les librairies à utiliser pour l'implémentation du client. Finalement, à travers toutes les étapes de la phase d'implémentation, nous devons consulter différentes équipes de développement d'outils de visualisation. L'objectif de nos travaux est de proposer une solution générique et nous ne voulons pas avoir de biais dans la conception de la nouvelle architecture. C'est la raison pour laquelle nous cherchons des avis externes.

À la fin de la phase d'implémentation et réingénierie, les contributions à Trace Compass seront ajoutées à la branche principale du projet et nous proposons un prototype du client.

3.3 Évaluation de la solution

L'évaluation de la solution passe par deux critères. Le premier est fonctionnel, nous devons nous assurer que les vues affichées par le client web sont semblables aux vues correspondantes de Trace Compass. Pour ce faire, il s'agit simplement de comparer deux fenêtres côte à côte. Le deuxième critère est la performance. Nous devons écrire des tests de performance du côté client afin d'évaluer le surcoût associé à notre solution. Ces tests seront exécutés sur l'environnement `localhost` afin de ne pas prendre en compte la latence du réseau. Par ailleurs, nous utilisons des traces de l'ordre du gigaoctet, que nous avons générées en traçant la machine de test avec LTTng, afin de valider la mise à l'échelle de notre approche. Les configurations matérielles de la machine de test sont :

— Intel Core i7-6700 @3.40 GHz
— 16 Go de mémoire vive DDR4 @2133 MHz
— SSD de 120 Go de Sandisk

Le système d'exploitation utilisé est Ubuntu 16.04.03 LTS avec le noyau 4.13.0-41-generic. Les tests de performance du client s'exécutent sur la version officielle de Chrome 65 et le serveur Trace Compass s'exécute avec OpenJDK 1.8.0_171. Les tests de performance utilisent l'outil `performance.now()` de JavaScript pour mesurer le temps avec une précision de 5 microsecondes.

CHAPITRE 4 ARTICLE 1 : DISTRIBUTED ARCHITECTURE FOR INTEGRATED DEVELOPMENT ENVIRONMENT, LARGE TRACE ANALYSIS AND VISUALIZATION

Authors

Yonni Chen Kuang Piao <yonni.chen-kuang-piao@polymtl.ca>
Michel Dagenais <michel.dagenais@polymtl.ca>
École Polytechnique de Montréal

Soumis à Software : Practice and Experience

Abstract

Integrated development environments (IDE) provide many useful tools such as a code editor, a compiler and a debugger for creating software. Those tools are highly sophisticated and their development requires a significant effort. Traditionally, an IDE supports different programming languages via plugins that are not reusable in other IDEs. Given the high complexity and constant evolution of popular programming languages, such as C++ and even Java, the effort to update those plugins has become unbearable. Thus, recent work aims to modularise IDEs and reuse the existing parser implementation directly in compilers. However, when IDE debugging tools are not sufficient to detect performance defects in large and multi-threaded systems, developers must use tracing and trace visualization tools in their software development process. Those tools are often standalone applications and do not interoperate with the new modular IDEs, thus losing the power and the benefits of many features provided by the IDE. The structure and use cases of tracing tools, with the potentially huge execution traces, significantly differ from the other tools in IDEs. It is therefore a considerable challenge, not addressed until now, to integrate them to the new modular IDEs. In this paper, we propose an efficient modular client-server architecture for trace analysis and visualization that solves those problems. The experimental evaluation demonstrates that our proposed flexible and reusable solution is scalable and has a small acceptable performance overhead compared to the standalone approach.

4.1 Introduction

Creating software requires a set of development tools, such as a code editor, a compiler, a debugger and a profiler, that are often provided by an integrated development environment (IDE). Some popular programming languages such as C++, and even Java, evolve rapidly and have a very complex syntax. Thus, developing a high-quality compiler or debugger needs a significant effort. Traditionally, modern IDEs support multiple programming languages via plugins, but those plugins are tightly coupled to the IDE and hardly reusable in other IDEs (Keidel et al., 2016). It means that if m IDEs support n different languages, $m \times n$ plugins must be implemented. In recent times, upon new language versions, updating C++ and Java plugins, for widely used IDEs such as Eclipse CDT, in addition to updating the compilers, has become an unbearable effort. Recent work aims to change the traditional plugin approach into a language server approach, thus reducing the number of different implementations to $m + n$ (Keidel et al., 2016). A language server can be considered as an independent module that provides support for code completion, code errors, syntax highlighting, and other IDE services, through a standard protocol. Moreover, the parser implementation in the language compiler is directly reused as a language server.

However, when those development tools are not sufficient to detect performance defects on large multi-threaded systems, developers use tracing techniques to collect data during the software execution. The data collected is eventually written to disk into a trace file and its size can range from a few megabytes to hundreds of gigabytes. Therefore, analyzing manually the trace is almost impossible and automated trace visualization tools are required. As the software grows and becomes more complex, the trace visualization tools must be part of the IDE, like the debugger. For example, Trace Compass is an Eclipse-based trace visualization tool that integrates well with the Eclipse CDT IDE. Among other features, when looking at events in a trace, the source code location of the tracepoint emitting the event can be shown in the IDE.

The current trend towards modular IDEs and language servers is well established (Efftinge and Kosyakov, 2017; Microsoft). However, the use cases for tracing are significantly different from other IDE tools. Indeed, editors, compilers, and even debuggers are centered around the well defined and structured software package architecture (functions, files, libraries, applications, etc.). In tracing, event streams from several concurrent executions (parallel threads, interrupts, signals, user, and kernel space execution) are multiplexed and can create large trace files of possibly hundreds of gigabytes. Thus, the modularization of the tracing tools in an IDE presents very significant challenges, regarding data communication and where to perform the computation, which had not been addressed until now.

The recent evolution in software systems also introduces many desirable features or requirements for a new trace analysis architecture. The sources of tracing data are becoming numerous (user space, kernel space, hardware supported tracing, GPGPU co-processors, etc.) and each has specific formats, use cases, analyses and other particular features. Most of the time, the developer then needs to install several different tools which cannot interoperate and combine their information. In addition, trace analysis tools may operate in different contexts : interactive graphical user interface to investigate a local trace file, the same interface to investigate remote trace files from several interacting cloud nodes, an automated anomaly detection module using machine learning to monitor online traces from several cloud nodes, or even a continuous integration framework using trace files to verify the performance of new versions of a software package.

Those new requirements motivated us to change how we think about software runtime analysis, and to propose a new scalable and efficient IDE-integrated architecture for runtime software tracing, visualization, and analysis. We propose a client-server architecture that includes a back-end server for trace collection and analysis, an IDE pluggable client to show the analysis results and to correlate with the source code, and a neutral data transmission protocol between the server and client to ensure loose coupling (Trace Analysis Server Protocol). Thus, our main contributions, to address the current situation, where trace visualization tools do not support a number of important new use cases, and cannot integrate well with the new modular IDEs, are :

1. A new scalable and efficient modular client-server architecture for large trace analysis ;
2. The collaboratively developed Trace Analysis Server Protocol ;
3. A prototype implementation of this proposed architecture ;
4. A quantitative evaluation of the scalability, efficiency and overhead of the proposed implementation ;

To our knowledge, this is the first trace visualization tool that addresses the challenges brought forward by the recent modularization of IDEs, with a frontend user interface and backend servers for language parsing, debugging and now tracing.

This paper is organized as follows. First, we review the related work and the existing trace visualization tools in section 4.2. Then, we present the specification of the proposed architecture and detail its implementation in section 4.3. In section 4.4, we evaluate the proposed solution by measuring the data transferred, the performance overhead and the scalability. Finally, we conclude and present possible future work.

4.2 Related Work

The related work is divided into three sections. The first section will report recent works on IDEs. The second section will cover tracing and trace analysis from a high-level perspective. The third will present existing trace visualization tools and their architecture.

4.2.1 Recent works on IDEs

Microsoft's language server protocol (LSP) is a communication protocol, based on JSON-RPC, between a client, which is the IDE, and a server that offers language support (Keidel et al., 2016). At this time, the LSP does not specify how the messages exchanged should be transferred and the client is responsible for managing the server's lifetime. The protocol provides common features such as code completion, code error, syntax highlighting and go-to definition. Many organizations such as the Eclipse Foundation, Github, and JetBrains are adapting their popular IDE (Eclipse, Atom, IntelliJ) to implement the LSP.

Keidel et al. have presented *Monto* (Keidel et al., 2016) which follows the same idea as the LSP, but their approach allows the language server to be stateless. Services are responsible for providing the common language features, and those services may also be composed of smaller services. Moreover, in comparison to the LSP, their solution doesn't have to maintain and update a copy of the source code.

Marr et al. have presented the *Kómpos protocol* (Marr et al., 2017) which is a concurrency-agnostic debugger protocol. Its goal is to decouple the debugger from the concurrency models such as threads, locks, communication event loops, and others. The protocol provides support for common features such as breakpoints, step-by-step and visualization of the interaction of concurrent models. In comparison to existing debugger protocol such as Java Debug Wire Protocol or the GDB machine interface, their solution is not specific to a concurrency concept.

Efftinge and Kosyakov have presented *Theia* (Efftinge and Kosyakov, 2017), a new open-source IDE framework for building IDEs that could run both as a desktop application or in a web browser connected to a remote backend. The project shares a lot of similarities with Microsoft's code editor, Visual Studio Code. Theia uses Node.js, Electron and implements the LSP.

4.2.2 Tracing and trace analysis

Tracing is a performance analysis technique that records runtime events on an executing program. Before collecting those events, tracepoints must be inserted either statically or

dynamically. In the first case, the code must be modified to include tracing macros and must be recompiled. In the second case, tracepoints are added dynamically to a compiled and running program (Gregg and Mauro, 2011). At runtime, events will be emitted, and a program called tracer will capture them to eventually produce a trace file, organized in a specific format. Tracing can be done on several levels including user-space, kernel, hardware, hypervisor, network, etc. Given that tracing is used to detect and identify performance issues, a tracer must have a minimal execution overhead. A trace analysis transforms merely the trace events into states and reorganizes them into a tree structure, for faster access (Prieur-Drevon et al., 2018). Indeed, trace files could easily contain millions, even billions of events and the analysis must use an efficient data structure to maintain query performance.

4.2.3 Trace visualization tools

There are lots of standalone tracing tools which support different solutions of tracing and trace analysis. Ezzati-Jivan and Dagenais presented most of them in a survey (Ezzati-Jivan and Dagenais, 2017). However, in this section, we only review the recent open-source tools that can support multiple trace formats, large traces and user-defined analysis and some communications.

LTTng Scope (project) is a JavaFX-based desktop trace viewer that focus only on CTF (Desnoyers, 2012) traces produced by the LTTng tracer (Desnoyers and R Dagenais, 2006). The architecture is well defined, separating the trace analysis in a reusable library, called Jabberwocky, and the visualization part in another component. The trace analysis library supports large traces and exposes an API that the visualization component invokes directly. However, *LTTng Scope* does not support user-defined analysis and other trace formats.

TraceCompass is an Eclipse-based desktop trace visualization tool. It supports different trace formats and offers many comprehensive analyses (Prieur-Drevon et al., 2018). In his previous work, Kouamé has already studied and implemented an XML-based language for user-defined analysis in TraceCompass (Kouamé, 2015). This tool supports large traces, but its architecture is monolithic and not well defined ; some components from the views are tightly coupled to components from the analysis while others are difficult to maintain.

Google has its tracing infrastructure system within Chromium (Prieur-Drevon et al., 2018). To view the trace analysis results, a JavaScript front-end, called *Trace-Viewer*, have been developed. It supports the Trace Event Format (Duca and Sinclair) produced by Chrome Tracing and Android Systrace, but you can also write your trace format importer. The tool is accessible on Chromium, by navigating to `chrome://tracing`. Tracing, trace analysis and visualization are separates into different architectural layers. However, it does not support

large traces and user-defined analysis.

OpenZipkin (Strubel, 2017) is a distributed tracing system used to trace micro-services' architecture based systems created by Twitter. It's a tracer implementation of the OpenTracing specification, and Dapper inspires its design. The architecture is organized as a client-server architecture ; the backend is written in Java and using Spring Boot while the client is written in JavaScript. The client and the server are communicating through a custom HTTP REST API. This tool is limited to OpenTracing and does not support other trace formats. Moreover, it does not support large traces and user-defined analysis.

Jaeger (Technologies) is another tracer implementation of the OpenTracing specification by Uber Technologies. The architecture is organized as a client-server architecture where the server is written in Go, and the client is written in JavaScript. The client and the server are communicating through a custom HTTP REST API. Like OpenZipkin, this tool does not support other trace formats, large traces and user-defined analysis.

4.3 Proposed solution

Our proposed solution consists of a client-server architecture with multiple layers that have different roles and responsibilities. Figure 4.1 represents an overview of the proposed architecture and its components which will be explained in details in the following sections.

The architecture separates traces, trace analysis and clients. By applying the separation of concerns principles, each component of the architecture is independently maintainable. On the left, we have the element responsible for storing and managing traces that can be different sizes and in various formats (e.g., CTF, Trace Event format). Then we have the trace analysis server, responsible for providing specific analysis results on any input trace. Each server may offer different analyses or trace format support, so we need a mechanism to dispatch client requests to the right server. The API gateway pattern solves this problem (Richardson, 2017a). Besides, when we add or remove servers, we need a mechanism to update which analysis or feature is available, so we use the server-side service discovery pattern (Richardson, 2017b) to this end. Finally, we have the clients, responsible for consuming and showing the data produced by the server into useful views to the user. A client can be any system interested in showing trace analysis results whether it's an IDE, a monitoring system or a continuous integration system.

To support heterogeneous clients, we introduced the Trace Analysis Server Protocol (TASP), which requires TASP connectors in the clients and TASP services in the trace analysis server. We assume the heterogeneous nature of the clients because they may have different computing

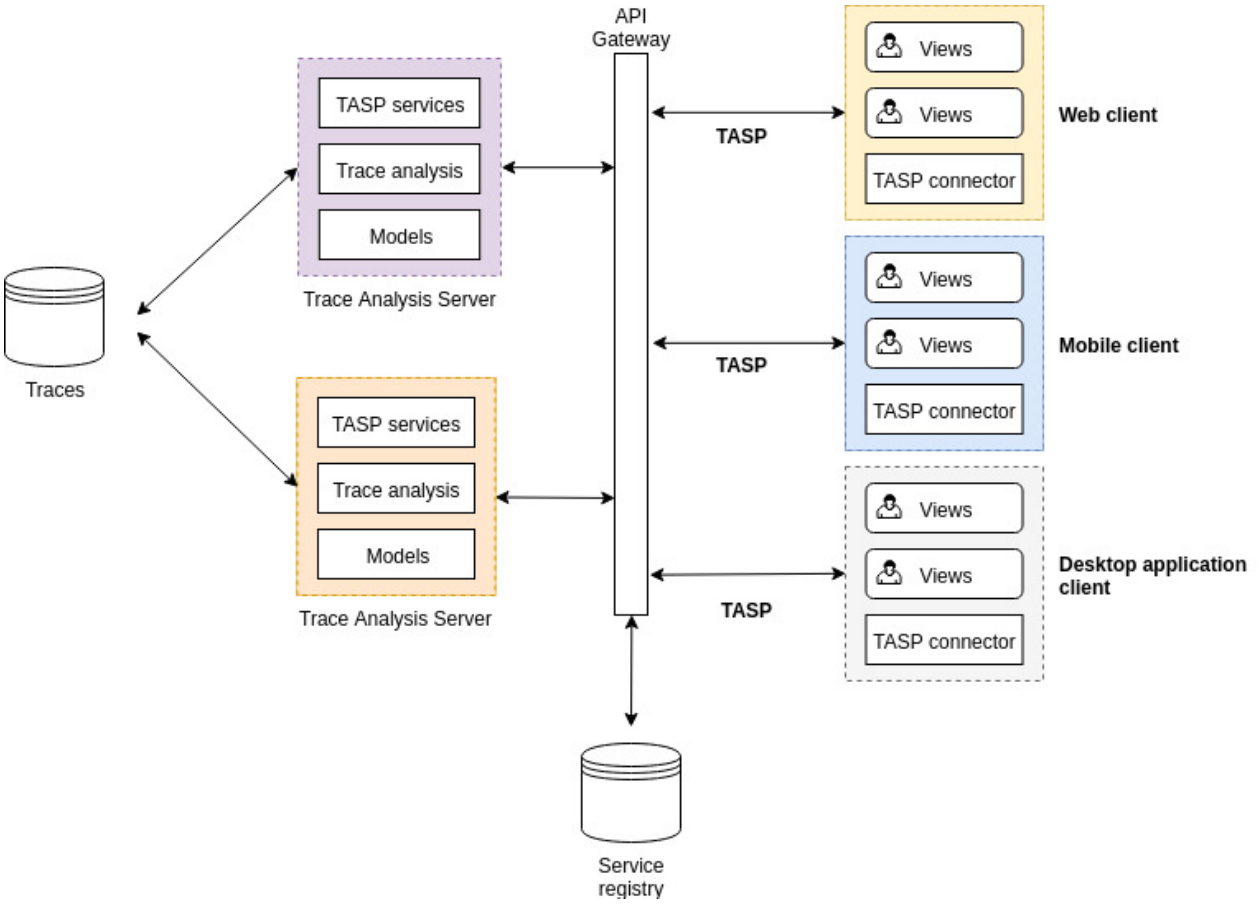

Figure 4.1 Client-server architecture for distributed trace analysis

power and may be written in different programming languages. Besides, the communication via the TASP is done over the network which allows the trace analysis server deployment to be in a cluster of high-performance computers. Therefore, our proposed architecture could profit from distributed systems, benefiting scalable computing power and resources while also having the trace analysis results available anywhere at any time without depending on a specific program installation to view them. If the deployment in the cloud is not the desired scenario, our solution can also be deployed locally.

Even though the server does the whole calculation workload, communication between the client and the server through the TASP could be expensive. Indeed, compared to a monolithic architecture, there is an execution time overhead related to setting up connection, to the network latency and the serialization/deserialization operations. The client also has an essential role in the architecture to reduce the number of requests to the server. For this reason, we will also detail the architecture on the client side by presenting its components,

the caching mechanisms and other techniques designed to this end.

4.3.1 The trace analysis server architecture

Wininger et al. explained, in their previous work, illustrated by Figure 4.2, the common trace analysis approach. A trace analysis reads a trace sequentially and stores its results (called state models) in a data structure that can be written on the disk. From the trace events, the analysis would extract the intervals of the state values for different system resources (CPU, processes, disks, etc.) (Montplaisir et al., 2013). Montplaisir et al. have already studied and proposed an efficient model to query such data structure. Then, views query this data structure, filter the relevant information and transform them into a high-level model to be displayed on charts.

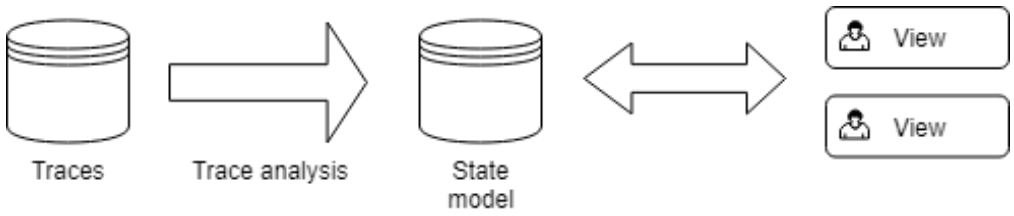

Figure 4.2 Common trace analysis approach

Any system interested in showing the analysis results would use a library that queries the state models. However, this approach has two limitations. First, if the involved systems are heterogeneous, the library must serialize the state model into an interoperable format such as the Extensible Markup Language (XML) or the JavaScript Object Notation (JSON). Prieur-Drevon et al. show that the data structure containing the state model can be massive (reaching gigabytes) depending on the trace. Therefore, serializing the state models directly is not efficient nor suitable if the data is transferred through the network. Second, the results of a query to the data structure containing the state models are not directly practicable, and the visualization systems must transform them. It means that every different system interested in showing a particular trace analysis result must implement its transformation of the state models before displaying it to the user. Those limitations motivated us to introduce new architectural components, shown in green in Figure 4.3, in the trace analysis process.

To have the smallest possible data serialization and avoid rewriting code to transform the state models, we use a Service-Oriented Architecture (SOA) approach. Our proposed trace analysis server architecture introduces the Data Provider component, responsible for querying the state models' data structure, filtering the results and transforming them into generic high-level models. Any system interested in showing the generic models queries the Service

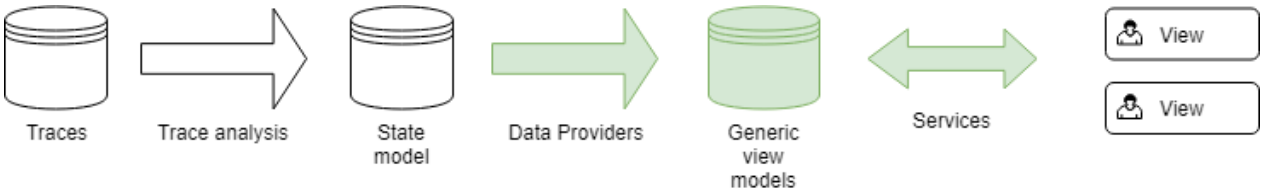

Figure 4.3 Proposed trace analaysis server architecture

component via the TASP. The Service component is responsible for implementing the TASP and serializing the model computed by the Data Provider component. The model received by those systems is practicable, and there is almost no modification needed to display it directly into charts.

Applying SOA's principles in the development of a system facilitate its scalability, evolvability, manageability and interoperability (Valipour et al., 2009). In the context of trace analysis, those qualities are necessary for supporting the addition or modification of new features, such as user-defined analysis, while reducing the cost of developing them. For those reasons, we chose an SOA approach for the trace analysis server, and its composition will be detailed below. Before deciding how clients would query the Service and the Data Provider components, we must determine *what* are they querying.

Extracting generic view models

The trace visualization tools presented earlier provide a lot of different useful views for displaying the analysis results. One of the most important types of view is time graphs. Figure 4.4 shows an example of a time graph view in TraceCompass. Other trace visualization tools such as Trace-Viewer or LTTng Scope provide the same type of view but within a different user interface.

On the left side of the view, we see a list of entries organized in a tree hierarchy. In practice, those entries are routines such as a process, a thread or a function that have a start time, an end time and a name. All time values could be represented as the Unix Epoch time in nanoseconds. On the right side of the view, we see a Gantt-like chart that shows rectangles of different colors, which we will call *states*. These states are bounded by a start time and an end time. Each time graph routine on the left is associated with a collection of states on the right, that we call a row. Each state has a value, which the view maps to a rectangle color, and may also have a label. Along with the Gantt-like chart, we may have arrows, starting from a state of a routine to another state of another routine. Those elements are common

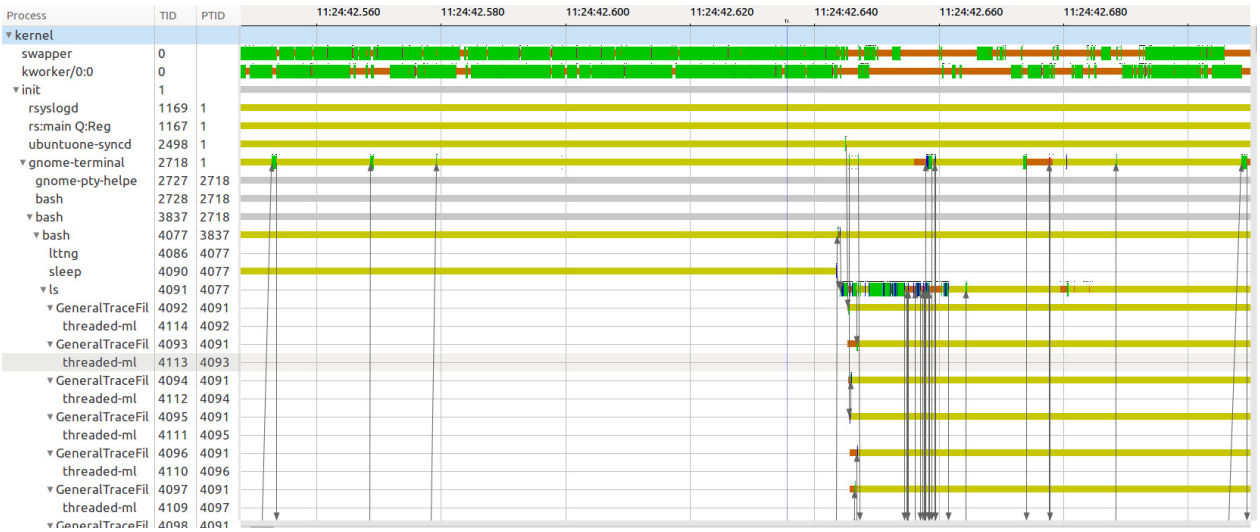

Figure 4.4 The thread status view in TraceCompass is an example of a time graph view

to all trace visualization tools presented. In practice, time graph views are used to show the results of the critical path analysis, the flame graphs analysis or the call stacks analysis. From the previous elements, we extracted common models for time graphs views :

Code 4.1 Time graph models definition

```
typedef long timegraph_routine_id;

struct TimegraphRoutine {
    timegraph_routine_id id;
    timegraph_routine_id parentId;
    long startTime;
    long endTime;
    string name;
}

struct TimegraphState {
    long startTime;
    long endTime;
    int value;
    string label;
}

struct TimegraphRow {
```

```
    timegraph_routine_id routineId;
    TimegraphState[] states;
}

struct TimegraphArrow {
    timegraph_routine_id sourceId;
    timegraph_routine_id destinationId;
    long startTime;
    long endTime;
}
```

The second important type of view that we focus on is XY views. An XY view display on a chart one or multiple series, which is merely a mapping between a collection of x and y values. Each series on the chart has a name. Those views can be shown as a bar chart, as a line chart or as a scatter chart. In practice, XY views are used to display the results over time of the CPU usage analysis, the memory usage analysis, the disks I/O activities analysis or the counters analysis. As an additional feature, Trace Compass also display on the left side of the view a list of selectable entries used for filtering series. For example, the CPU usage view in Trace Compass displays processes and threads on the left, allowing the user to filter the CPU usage per process. The filtered series then appear on the right side so that you can compare them easily. Considering that, the model extracted is :

Code 4.2 XY models definition

```
struct XYSeries {
    long id;
    string name;
    long[] x;
    double[] y;
}

struct XYModel {
    string name;
    XYSeries[] series;
}
```

These models were designed to be generic and compact. By being generic, it can be created either by LTTng Scope, Trace-Viewer or Trace Compass. Moreover, the model applies to any new trace analysis implementation which results can be displayed either in a time graph view

or an XY view. By being compact, only the smallest relevant information is serialized and transferred over the network to the client. Finally, the introduced models have no direct links with the state models, which decouples the trace analysis internal implementation from the clients.

The Data Providers layer

This layer of abstraction is used as an interface to build and fetch the generic models presented earlier. Figure 4.5 shows the class diagram of the Data Provider layer. Therefore, we have different interfaces for each model : one for fetching trees-like models, one for fetching XY models and one for fetching time graph models. Implementation of any of these interfaces is a class that gives the results of an analysis.

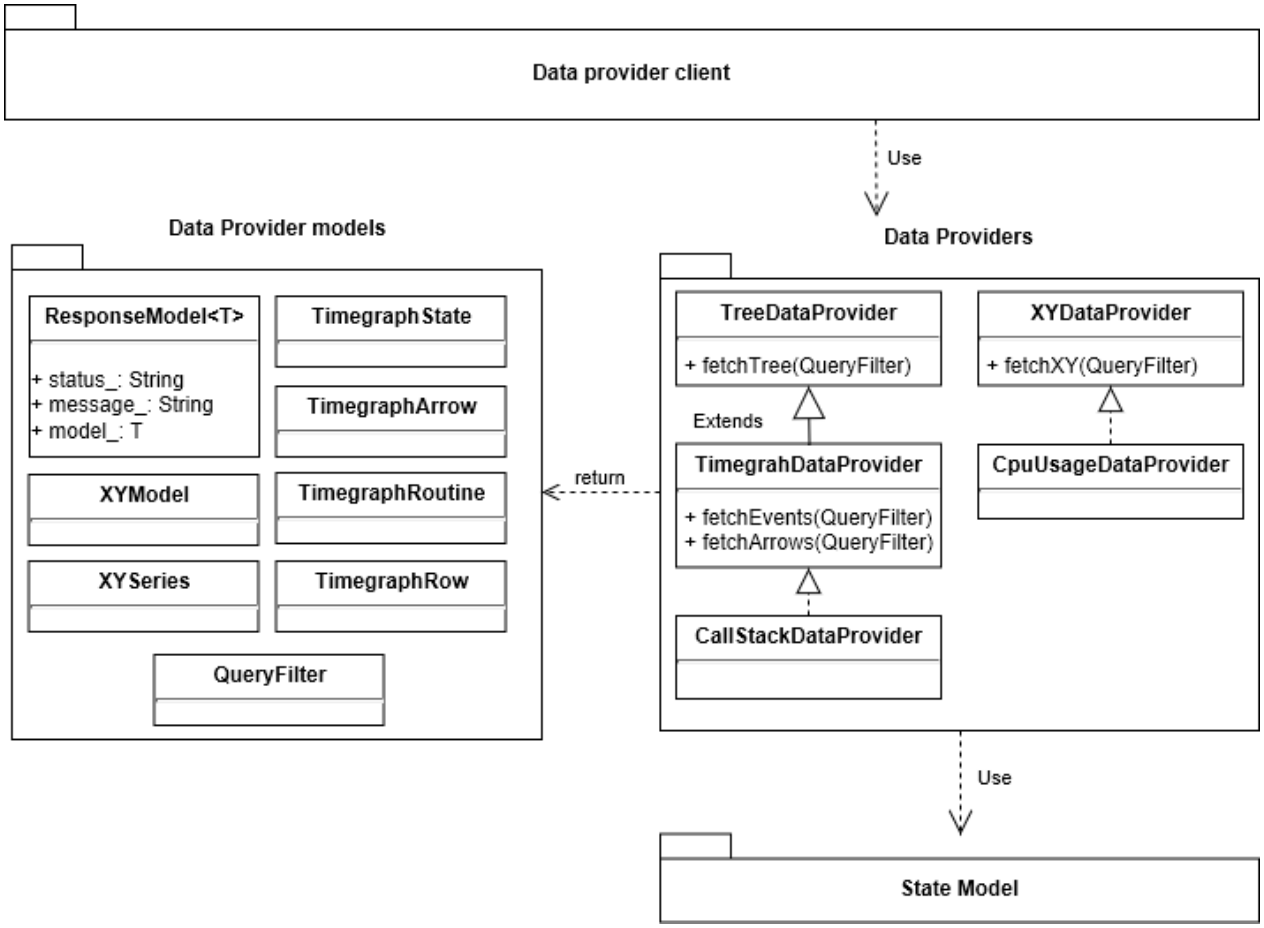

Figure 4.5 Class diagram of the Data provider layer

The Data Provider interfaces also provide a mechanism for fetching a specific model given

a `QueryFilter` defined below. In practice, we are interested in models bounded in a particular interval of time ; When the user zooms in or out, pans left or right, the bounds in the `QueryFilter` must change accordingly. Furthermore, depending on the resolution of the computer screen, we may be interested in showing a lower or higher definition of the model. Finally, we may be interested in particular data entries that have an id such as the `TimegraphRoutine` or the `XYSeries`. Thus, the Data Provider must allow fetching only them. This mechanism gives a lot of flexibility to the Data Provider client, but the primary motivation is to minimize the amount of data to transfer and the serialization overhead. If the client is only interested in querying a specific part of the model, there is no need to return the results for the whole analysis. However, it is still the client's responsibility to optimize queries to the data provider.

Code 4.3 QueryFilter definition

```
struct QueryFilter {
    long start;
    long end;
    int count;
    long[] selectedIds;
}
```

The Data Provider layer is also designed to support non-blocking fetch operations. Indeed, for large traces, the analysis could be very time-consuming. For this reason, the fetch method returns a `ModelResponse` that encapsulates a model, a status, and a status message. The status is simply an *enum*, mapping integers with a state : *running*, *completed*, $failed$ or *canceled*. When we get a $failed$ status, we may be interested to know why. That's why the `ModelResponse` has a status message, a string detailing the status state.

For example, assume that a Data Provider client is interested in a time graph model from t_1 to t_{10}. When receiving the request, the data provider may not have the analysis result up to t_{10} but to t_5 only. Instead of waiting for the analysis to complete, the data provider will return a `ModelResponse` with *running* as status and the model associated with $[t_1, t_5]$. It is the client's responsibility to request the data provider as long as a *completed* status is not received.

The Service layer

While the Data Provider layer provides a simple API for fetching models, the service layer is responsible for serializing them and for providing interoperability with the clients. This

layer can be considered as a facade of the trace server analysis that implements the TASP. To ensure interoperability, many mechanisms exist such as remote procedure calls (RPC) or object request brokering (e.g., CORBA) (Wileden and Kaplan, 1999). The trace analysis server and the client must use the same mechanism, so the chosen approach must be implemented in different programming languages. Finally, the Service layer is highly coupled to the TASP specification which means that the service implementation is influenced by the network protocol used for the TASP.

4.3.2 The Trace Analysis Server Protocol

The Trace Analysis Server Protocol (TASP) is profoundly influenced by the Language Server Protocol (LSP). As shown in Figure 4.6, its goal is to standardize how multiple development tools or clients communicate with trace analysis servers. Thus, any IDE that implements the TASP could interoperate with a trace analysis server and shows the same views provided by the standalone trace visualization tools. Moreover, the TASP makes trace servers interchangeable, and their implementation is abstracted for the client.

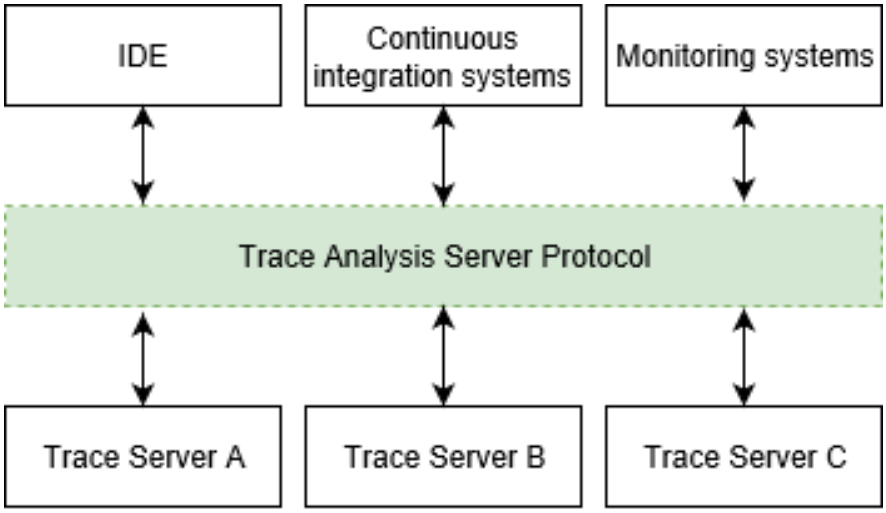

Figure 4.6 Interaction relying on the Trace Server Analysis Protocol

However, the LSP relies on the JSON-RPC protocol (Group et al., 2013) while we chose an HTTP Representational State Transfer (REST) approach. Both JSON-RPC and HTTP protocols are stateless and implemented in the most popular languages, but they don't provide the same flexibility in the serialization format. JSON-RPC uses only JSON format which could be suboptimal regarding data size (Sumaray and Makki, 2012). Serializing the models into a text-based format could easily reach megabytes, and the TASP should serialize them

into a binary-based format to minimize the data transferred. Using compression (GZip) over JSON could potentially reduce the size of the data, but this approach implies using more CPU and leads to more execution time overhead. HTTP provides a mechanism to send binary data by setting the HTTP response header `Content-Type` to `application/octet-stream`. We chose Google's Protocol Buffers, also called Protobuf (Google, 2018), as the binary format to serialize the models.

Masse have presented a guide for designing consistent RESTful APIs (Masse, 2011) that we followed for the specification of the TASP. Its full specification is available on Theia IDE's Github page (Ericsson). The TASP design is a result of collaborative work between the EffiOS' team working on *LTTng Scope*, the Ericsson's team working on *TraceCompass* and our research lab. The goal of this collaborative work was to propose a protocol that would be the least specific to the inner workings of a particular trace analysis tool.

4.3.3 The client architecture

The proposed client architecture has three principal components : TASP Connector, Widget, and Chart. The TASP connector is responsible for querying the server and for returning a view model entity. The Chart gets a view model and is responsible for drawing it and displaying it. Finally, the Widget is acting like a controller. Because the TASP Connector and the Chart don't know each other, the Widget is responsible for calling the TASP Connector's methods and feeding the returned model to the Chart. As we explained, communication between the TASP Connector of the client and the TASP services of the trace analysis server could be expensive, and the Widget component has an important role in reducing the requests sent to the server. Besides, to transfer as little data as possible, the client is also responsible for narrowing its request to the minimum. In this section, we will explain how the client can optimize its request to limit the data transferred, and we will present two techniques to reduce the number of requests to the trace analysis server : throttling and caching.

Reducing the size of transferred data

The client should know information about the screen size. Let's say that the client wants to display an XY view on a 1920x1080 pixels screen. The worst case is that the view is taking the whole screen, which means that we need to query one point per horizontal pixels. Thus, we don't need to query more points than the number of available pixels. The same idea goes for the time graph states and entries. We may have thousands of entries, but we may not have enough vertical pixels to show them. The given example was for a standard 1080p screen, but the client must be able to adapt to larger screen resolution such as 2K and 4K.

Throttling

Throttling is a technique for controlling the consumption of resources by an application (Homer et al., 2014). As we explained above, the client might receive an incomplete model when querying the server. As long as the completed status is not received, the TASP connector may make N consecutive requests to the server. For example, suppose that 10 seconds are needed to get the complete view model from t_1 to t_{100}, each request transfer 100 KB and that each request and redraw operation takes 200 ms. It means that the user interface is refreshed five times per second, we need to make 50 requests to the server until completion and we transfer 5 MB. Throttling helps us reducing this number if we *wait* a certain time x before re-triggering the request and redraw operations. Let's say that we chose to wait for 800 ms, and it means that we will need ten requests to the server instead of 50. Consequently, the user interface is refreshed only once per second and we transfer five times fewer data (1 MB instead of 5 MB). The longer the waiting time, the less request we do to the server. However, the drawback of this approach is that the user interface becomes less responsive.

Caching

The idea of this technique is that the client maintains a cache and looks into it before requesting the server. If the client finds the desired information in its cache, it does not need to query the server. This technique is used to limit communication between the client and the server, but it leads to more memory consumption for the client and a mechanism to manage the lifetime of the objects in the cache. Besides, to save requests, we also save time. Indeed, looking in a local cache does not have network latency concerns.

We used caching for the view models. Instead of querying only the view model that is to be displayed on screen immediately, we query more. For example, let's take the time graph use case. We are interested in showing the time graph row model of $t = [100, 200]$ for entries $e = \{3, 4, 5\}$. Instead, we will query the server for $t' = [75, 225]$ and $e' = \{1, 2, 3, 4, 5, 6, 7\}$ but will only display the data described by t and e. As shown in Figure 4.7, the green component is what the user sees, and we have the blue component in the cache. This approach can be seen as the opposite of the first suggested technique because we don't request the strict minimum, but both approaches complement each other. Indeed, a combination of both techniques leads to better results when the user is panning vertically or horizontally. For example, let's say that instead of requesting 100 KB of data, we query 140 KB because of caching. Every time that the user is panning in a way that what will be shown is available in the cache, we save a request of 100 KB.

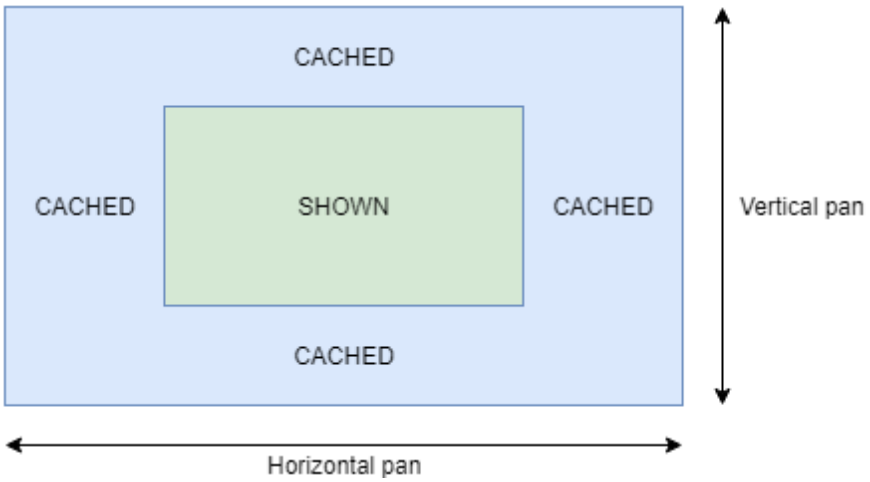

Figure 4.7 Vertical and horizontal caching for time graph

However, this technique does not work well when the user is zooming, because every zoom in or out operation leads inevitably to a new request. Thus, if the user is mostly zooming, this technique leads to worst results for the total of data transferred. For this reason, we suggest having a reasonable cache size, which is arbitrarily 25-50% larger than the shown area. Besides, caching is also useful for models that won't or won't likely change according to resolution or time interval. Let's take for example the disks I/O activity analysis. The list of drives won't probably vary according to the resolution or the time intervals. Thus, the model can be requested only once and stored in the cache.

4.4 Evaluation

In this section, we focus on evaluating the proposed solution. To this end, we implemented the client-server architecture. While the proposed architecture solves the initial problem, we want to ensure, for a wide range of trace sizes, that the data transferred is small and the performance overhead is acceptable in comparison to the standalone trace analysis tool.

4.4.1 Implementation

A full implementation of the server architecture is available in the Trace Compass main project and incubator project. The Data Provider implementation is on the master branch of the main project, while the HTTP services implementation is on the master branch of the incubator project :

— `https://git.eclipse.org/r/#/q/project:tracecompass/org.eclipse.tracecompass`

— `https://git.eclipse.org/r/#/q/project:tracecompass.incubator/org.eclipse.tracecompass.incubator`

We also implemented the client architecture, written in JavaScript and running within a web browser, which will query Trace Compass through the TASP. This implementation, named TraceScape, is available as an open source project in the following GitHub repository : `https://github.com/cheninator/trace-scape`.

4.4.2 Tests environment

All the tests were executed on *localhost* on a machine with an Intel Core i7-6700 @ 3.40 GHz, 16 GB RAM and running Ubuntu 16.04 LTS with the kernel 4.13.0-41-generic. The web client uses the official build of Chrome 65, and the Trace Compass server runs on OpenJDK version 1.8.0_171. A trace of 2.47 GB was generated by tracing all kernel events of the test machine with LTTng 2.10.3. The trace contains 1580 threads and around 75 million events. The generated trace is an ideal candidate since we want to show that our solution supports large traces (size on disk higher than 1 GB). After completion of the thread status analysis and the kernel memory usage analysis, their respective state model size on disk are 1.9 GB and 212 MB. The trace and the state models are stored on a 120 GB SanDisk SSD.

In the tests conducted to measure the data transferred and the execution time overhead, the client sends requests to the server, for the analysis results from the start to the end of the trace, with realistic values for the resolution. Moreover, the client sends requests for different models ; the `XYSeries` for the kernel memory usage analysis and the `TimegraphRow` for the thread status analysis. For consistency, we make sure to request the same collection of `TimegraphRow` and `XYSeries`.

Three factors can influence the data transferred and the execution time overhead : the desired resolution of the model, the number of different `TimegraphRow` or `XYSeries`, and the size of the trace. In each test, we fixed two out of these three factors and varied the other. For each test result, the full-line series are the results of the average of 10 executions, and the dotted line series is the average plus or minus the standard deviation.

4.4.3 Data transferred

The quantity of data to be transferred during a single session depends greatly on how many requests are done from the client to the server. Instead of evaluating it, we chose to measure the data transferred per request. We compare three serialization formats : JSON, JSON compression with GZip and Protobuf.

XY models

In the first test, we requested only one XY series, and we changed the desired resolution (i.e., the number of points). Figure 4.8 shows the results for the first test. In the second test, we fixed the resolution to 1000 and changed the number of different XY series. Figure 4.9 shows the results for the second test.

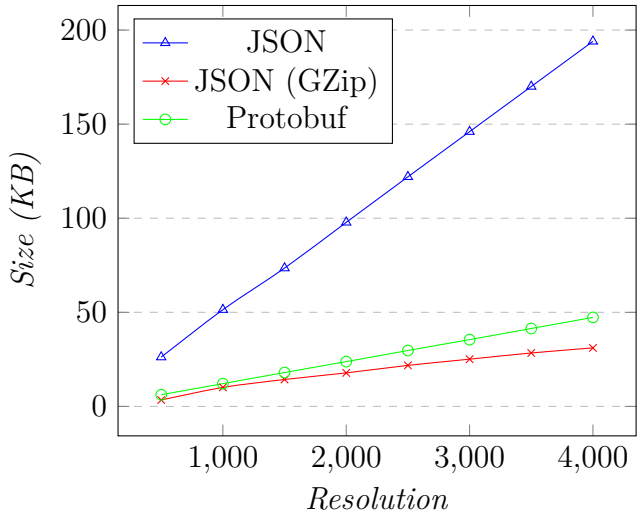

Figure 4.8 Comparison of the amount of data transferred for requesting the XY model. Fixed number of XY series to 1, changing the resolution.

As the resolution is increasing, the amount of the data transferred over Protobuf and compressed JSON are close. Compared to JSON, using compression gives around 80-87% of space saving while Protobuf is more constant, with 75-77% of space saving. Finally, for 4000 points, which is near a 4K monitor, we transferred less than 50 KB per request for both Protobuf and compressed JSON while nearly 200 KB is needed for JSON.

However, as the number of XY series is increasing, using compression is a lot more efficient than Protobuf. Figure 4.9 shows that GZip is better than Protobuf in this test. Compared to JSON, compression gives around 80-97% of space saving while Protobuf gives around 59-76%. Finally, when we request 35 XY series, we need to transfer less than 30 KB for compressed JSON while approximately 420 KB is necessary for Protobuf.

Time graph models

We did similar tests for the time graphs models. In the first test, we fixed the number of different time graph rows to 25, and we changed the resolution. Figure 4.10 shows the results

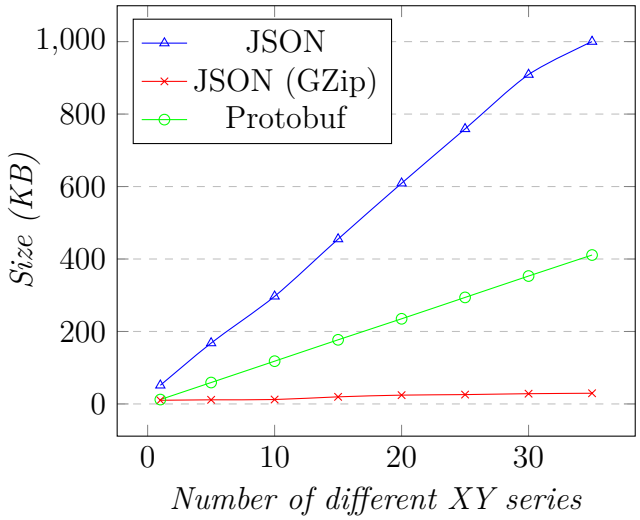

Figure 4.9 Comparison of the amount of data transferred for requesting the XY model. Fixed resolution to 1000, changing the number of different XY series.

for the first test. In the second test, we fixed the resolution to 1000, and we changed the number of different time graph rows. Figure 4.11 shows its results.

Like XY models, as the resolution is increasing, the amount of the data transferred over Protobuf and compressed JSON are close. Compared to JSON, using compression gives around 90% of space saving while Protobuf gives around 84% of space saving. Finally, for a resolution of 4000, we transferred less than 500 KB per request for compressed JSON, nearly 650 KB for Protobuf and over 4 MB for JSON.

However, unlike XY models, compressed JSON is not significantly more efficient than Protobuf as the number of time graph rows increases. If we compared to JSON, space savings are around 88-89% and approximately 84% respectively for compressed JSON and Protobuf. Finally, when we request 200 time graph rows, about 500 KB is transferred for Protobuf, less than 350 KB for compressed JSON and 3 MB for JSON.

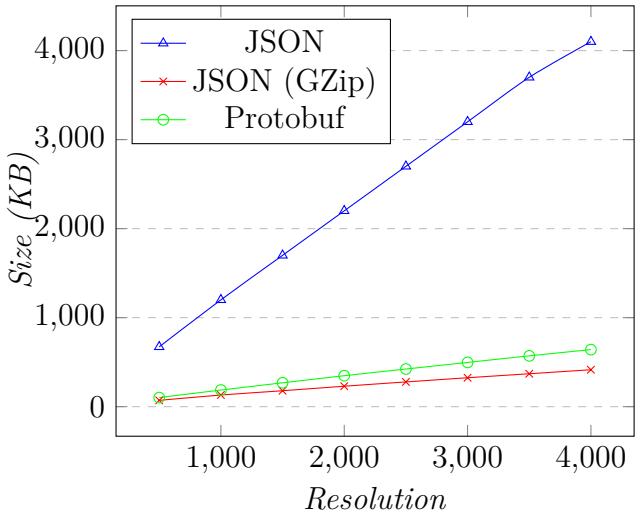

Figure 4.10 Comparison of the amount of data transferred for requesting the time graph row models. Fixed number of time graph row to 25, changing the resolution.

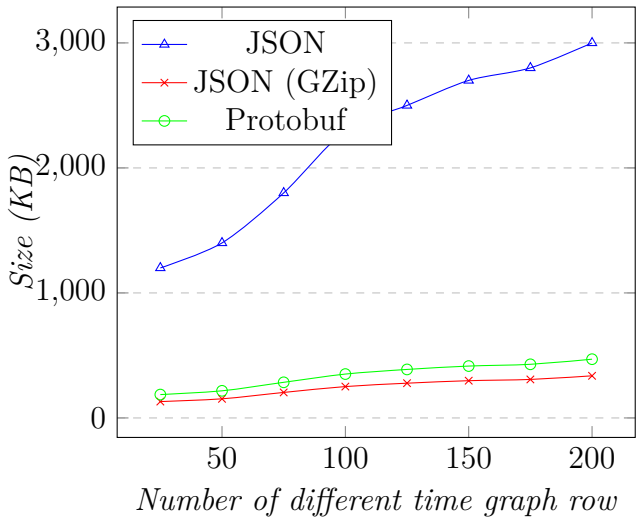

Figure 4.11 Comparison of the amount of data transferred for requesting the time graph row models. Fixed resolution to 1000, changing the number of time graph rows.

4.4.4 Execution time overhead

The client-server architecture introduces an execution time overhead. Given that the tests are run on *localhost*, this overhead is mainly composed of opening/closing TCP connections, sending/receiving HTTP headers and serializing/deserializing the data. The network latency must be considered when we are not in a *localhost* environment. The execution time overhead is given by :

$$Overhead(ms) = T_{TCP} + T_{Headers} + T_{Serialization} \tag{4.1}$$

To get the view models, the client makes an HTTP request to the server, which queries the Data Providers. Thus, the time for completing a request from the client to the server is the sum of the time for querying the Data Provider and the execution time overhead :

$$T_{Request} = Overhead(ms) + T_{Query} \tag{4.2}$$

On the web client, we measured the time for completing a request to the server. On the server, we measured the time for querying the Data Provider. To get the execution time overhead, we subtract the two measured execution times. Besides having the execution time overhead in ms, we are also interested in getting the execution time overhead in percentage :

$$Overhead(\%) = \frac{Overhead(ms)}{T_{Query}} * 100 \tag{4.3}$$

Given that the serialization time depends significantly on the size of the model, we show in this test how the execution time overhead varies depending on the model and its size.

XY models

We reused the same test configurations presented in the previous section, but for measuring the execution time overhead instead of the amount of data transferred. Figure 4.12 shows the results of the first test and Figure 4.13 shows the results of the second test.

As the resolution increases, the execution time overhead in ms increases as well, which is expected because the size of the model affects the serialization time. However, the cost for Protobuf seems fairly constant for the tested resolutions. There is indeed a fixed cost for setting up the connection and thereafter the serialization with Protobuf is extremely efficient. Unsurprisingly, using compression has more execution time overhead than using

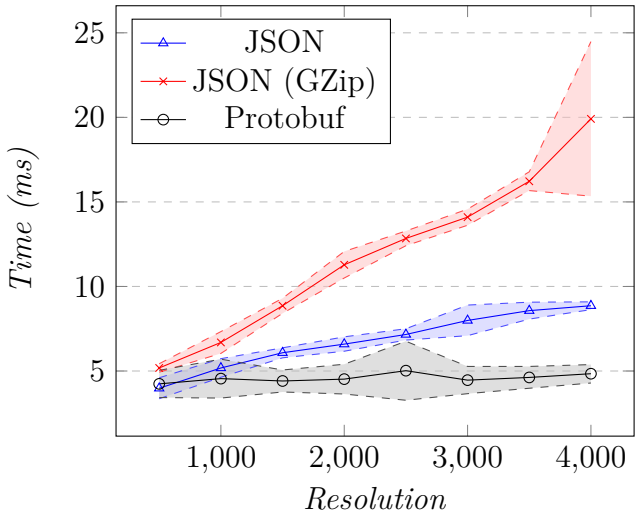

Figure 4.12 Comparison of the execution time overhead for requesting the XY model. Fixed number of XY series to 1, changing the resolution.

JSON. Finally, for 4000 points, there is around 5 *ms* overhead for Protobuf while it reaches 20 *ms* for compressed JSON and 9 *ms* for JSON. Table 4.1 shows the overhead in percentage.

Table 4.1 Relative execution overhead for requesting the XY model for 1 series

	Overhead (%)		
Resolution	JSON	JSON (GZip)	Protobuf
500	7.1	11.4	9.9
1000	5.2	7.8	4.8
1500	4.6	6.7	3.2
2000	3.8	6.6	2.7
2500	3.8	6.5	2.7
3000	3.7	6.2	2.0
3500	3.7	6.5	1.9
4000	3.4	7.3	1.8

Compressed JSON leads to 11.4% overhead for 500 points and 7.3% overhead for 4000 points while Protobuf leads to 9.9% and 1.8% respectively. In between, we have 7.1% and 3.4% overhead by using JSON directly. Those results show that when we request more points, the trace analysis server spends more time to compute the model than to serialize it.

As the number of XY series is increasing, the execution time overhead in *ms* increases as well. For 35 XY series, it reaches nearly 50 *ms* for compressed JSON, 40 *ms* for JSON and 14 *ms* for Protobuf. Even if those times seem to be small, the percentage, on the other hand,

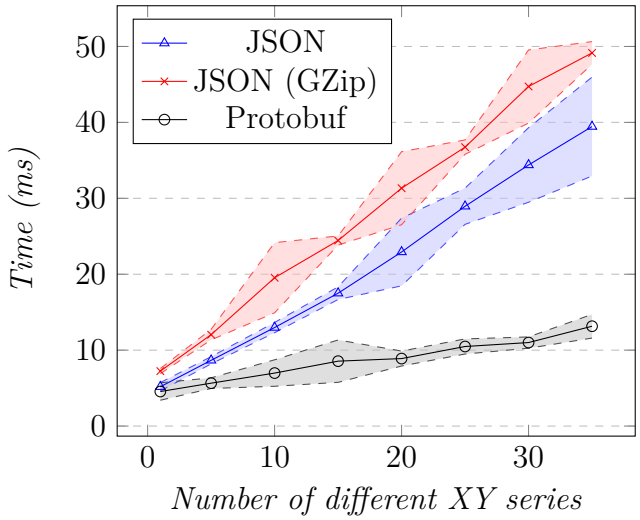

Figure 4.13 Comparison of the execution time overhead for requesting the XY model. Fixed resolution to 1000, varying the number of different XY series.

is high. Table 4.2 shows the overhead in percentage.

Table 4.2 Relative execution overhead for requesting the XY model. Fixed resolution to 1000.

	Overhead (%)		
Number of XY series	JSON	JSON (GZip)	Protobuf
1	5.1	7.8	4.8
5	9.2	12.8	5.5
10	13.0	21.2	7.0
15	18.5	25.1	8.6
20	25.3	34.3	9.2
25	29.7	28.4	10.8
30	36.2	47.3	11.7
35	40.9	48.0	13.2

Using compression leads to the highest overhead for execution time. It reaches 7.8% for 1 XY series and 47.9% for 35. On the other hand, Protobuf has the smallest, leading to 4.8% overhead for 1 XY series and 13.2% overhead for 35. In between, we have respectively 5.1% and 40.9% overhead by using JSON. Those high percentages are explained by the fact that querying more entries from the Data Provider does not take much more time than serializing the computed result.

Timeline models

We measure the execution time overhead for the same test configurations as the previous section. Figure 4.14 shows the results for the first test. Figure 4.15 shows the results for the second test.

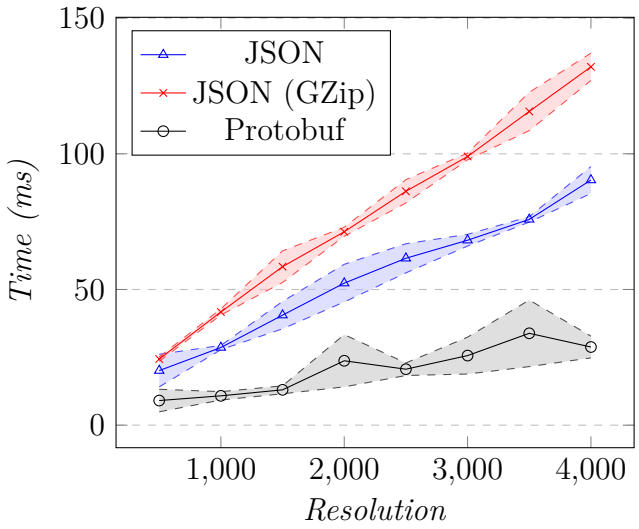

Figure 4.14 Comparison of the performance overhead for requesting time graph row models. Fixed number of time graph rows to 25, changing the resolution.

As the resolution is increased, the execution time overhead also increases, to exceed 130 *ms* for a resolution of 4000 points if using compression while it is nearly 30 *ms* for Protobuf. Table 4.3 presents the execution time overhead in percentage.

Table 4.3 Relative execution overhead for requesting the time graph row models. Fixed number of time graph rows to 25.

	Overhead (%)		
Resolution	JSON	JSON (GZip)	Protobuf
500	11.3	13.4	5.2
1000	9.6	13.3	3.5
1500	9.6	13.5	3.0
2000	9.5	12.5	4.2
2500	8.9	12.1	2.9
3000	8.0	11.4	3.0
3500	7.6	11.3	3.3
4000	7.8	11.2	2.4

Compressed JSON leads to the worst results, having a 13.4% overhead for a resolution of 500

and 11.2% for a resolution of 4000 points. Protobuf, on the other hand, scales well, having 5.2% and 2.4% overhead respectively. In between, we have 11.3% overhead for a resolution of 500 and 7.8% overhead for a resolution of 4000 by using JSON. The same observation as XY for models holds, when we request higher resolutions, the server spends more time computing the model than serializing it.

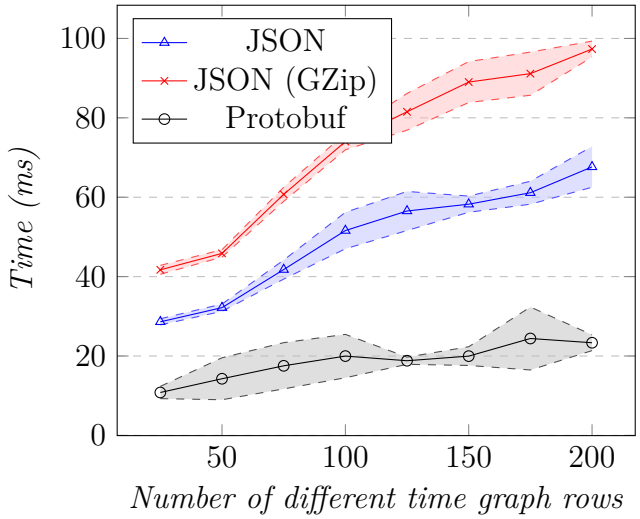

Figure 4.15 Comparison of the performance overhead for requesting the time graph row models. Fixed resolution to 1000, changing the number of time graph rows.

Like for XY models, as the number of time graph rows is increasing, using compression leads to the highest overhead for execution time, reaching almost 100 *ms* for a resolution of 4000 points. On the other hand, Protobuf has the smallest overhead, with less than 25 *ms*. Finally, table 4.4 shows the overhead in percentage.

Table 4.4 Relative execution overhead for requesting the time graph row models. Fixed resolution to 1000.

	Overhead (%)		
Number of time graph rows	JSON	JSON (GZip)	Protobuf
25	9.6	13.4	3.5
50	10.6	14.7	4.6
75	13.8	19.8	5.6
100	16.7	23.7	6.4
125	18.2	25.6	6.0
150	18.5	27.8	6.2
175	19.3	27.7	7.4
200	21.3	29.4	6.8

Using compression reaches 13.4% overhead for 25 entries and 29.4% for 200 entries. Protobuf has the smallest cost, leading to 3.5% for 25 time graph rows and 6.8% for 200. In between, we have respectively 9.6% and 21.3% for JSON. Also, as for XY models, when querying more time graph rows from the Data Provider, the server spends more time serializing the model than computing it.

4.4.5 Scalability

In this section, we evaluate how our proposed solution scales with the size of the trace. For each view model, we are interested in measuring how the size of the trace affects the data transferred and the execution time overhead per request. The traces used were also generated using LTTng. Their content in terms of the number of threads and number of events are not relevant for this test, we only take into consideration the size on disk.

XY models

The trace size should not have a significant impact on the amount of data transferred for XY models. In this test, we fixed the number of XY series to 1, the resolution to 1000 and we changed the trace size. In theory, given that we always query for a constant number of points and a constant number of series, the size of the data transferred should be constant. However, as shown in table 4.5, it is not necessarily the case for JSON and compressed JSON.

Table 4.5 Data transferred for requesting the XY model according to the trace size

	Data Transferred (KB)		
Trace size (MB)	JSON	JSON (GZip)	Protobuf
1.1	40.1	1.3	12.1
57.8	44.5	1.6	12.1
124.2	43.6	2.4	12.1
471.8	49.4	8.3	12.1
1507	49.2	10.3	12.1
1965	51.4	9.0	12.1
2470	51.4	10.1	12.1
3381	51.4	9.7	12.1

The reason why the amount of data transferred over Protobuf is constant is that each field of the model is serialized with a fixed number of bytes. The y property in the XY model is an array of *double* and Protobuf constantly uses 8 bytes to encode a *double* value. The x property is an array of *long* and Protobuf use *varints* to encode them. Varints are a method

for serializing an integer in one or more bytes depending on the value (leading null bytes are skipped). Since we use Unix Epoch times in nanoseconds in x, every value is large and Protobuf constantly uses 8 bytes. However, we were able to get a size reduction by using differential time values, which are much smaller, as follows :

$$\begin{aligned} x[0] &= \text{Unix Epoch start time in nanoseconds} \\ x[1] &= \text{Unix Epoch time value for } x[1] - x[0] \\ x[n] &= \text{Unix Epoch time value for } x[n] - x[n-1] \end{aligned} \tag{4.4}$$

For example, if we have originally $x = [1000010, 1000020, 1000031, 1000041]$, we would transfer $x = [1000010, 10, 11, 10]$. This technique leads to transferring fewer bytes. Indeed, when we send all the Unix Epoch times directly, we would transfer 17.0 KB for Protobuf instead of 12.1 KB. Nonetheless, as shown by the results, the amount of data transferred over Protobuf is still constant. The results may change if we insert something else in x. For JSON, each field of the model is serialized into a *string* where each character is encoded with 1 byte. Thus, the number of bytes needed to encode a *long* or a *double* value can vary. For compressed JSON, the size depends greatly on the data to be compressed. For example, if the y contains only 0s, compression would be optimal. Finally, the differences in term of data transferred are not significant compared to the difference between the trace sizes.

Table 4.6 Absolute execution time overhead for requesting the XY model according to the trace size

	Execution time (ms)		
Trace size (MB)	JSON	JSON (GZip)	Protobuf
1.1	5.25	5.8	5.34
57.8	5.11	6.3	4.14
124.2	4.76	5.74	5.16
471.8	4.83	6.24	4.75
1507	5.3	6.65	4.91
1965	4.86	6.41	5.05
2470	5.18	6.68	4.55
3381	5.08	7.31	6.09

The same observation holds for the execution time overhead. Given that we always receive a constant amount of data, the absolute execution time overhead in *ms* should be roughly constant. What varies is the percentage overhead. Indeed, as the size of the trace increases, the time required to query the Data Provider increases as well, simply because the size of the state model is bigger. This leads to a lower percentage overhead as the trace size increases. We

ran the same test as previously ; we fixed the number of XY series to 1, fixed the resolution to 1000 and changed the trace size. Table 4.6 shows the results.

Time graphs models

The size of the trace should affect the amount of the data transferred. A trace of a few megabytes does not contain the same number of events as a trace of few gigabytes. Thus, as the size of a trace increases, the number of states per time graph row increases as well, and this number influences the size of the data transferred. In this test, we fixed the number of time graph rows to 25, the resolution to 1000 and we changed the trace size. We compared the data transferred for each serialization format. Table 4.7 shows the results of the test.

Table 4.7 Data transferred for requesting the time graph row model according to the trace size

	Data Transferred (KB)		
Trace size (MB)	JSON	JSON (GZip)	Protobuf
1.1	143	11.5	21.3
57.8	215	22.6	32.7
124.2	862	87.9	126
471.8	1100	122	170
1507	1100	120	167
1965	1200	131	181
2470	1200	131	187
3381	1300	139	189

We observe that the difference in the amount of data transferred is significant between the 1 MB trace and the 500 MB trace. From 500 to 2500 MB, the increase is slower and seems to eventually reach a plateau. This can be explained by an optimization made by the Data Provider. If a state requires less than 1 pixel to be drawn, the Data Provider will not return it. We fixed the resolution to 1000 so, in the worst case scenario, we have 1000 states, which is one state per pixel.

As the amount of data transferred increases while the size of the trace increases, the execution time overhead should also increase. Like the data transferred, we should eventually reach a plateau. Table 4.8 shows the results of the execution time overhead. The same configuration as for the data transfer test is used. As expected, results show that a plateau is reached.

Table 4.8 Absolute execution time overhead for requesting the time graph row model according to the trace size

	Execution time (ms)		
Trace size (MB)	JSON	JSON (GZip)	Protobuf
1.1	6.34	7.76	6.16
57.8	7.54	10.0	7.07
124.2	20.27	29.03	9.80
471.8	26.30	38.73	10.31
1507	25.66	38.11	11.01
1965	28.72	39.58	10.86
2470	27.81	40.34	12.10
3381	29.22	42.35	11.03

4.4.6 Storage

The client-server architecture that we propose does not use additional disk storage. The view models are generated on-demand and are stored in RAM for caching, but are not written permanently on disk. For the local use-case, everything is working as before, the client and the server are running on the same machine and the server stores the trace and the state model on disk. However, for the distributed use-case, with the client and the server on different devices, this architecture guarantees that the client no longer needs to store the trace and the state model. Furthermore, the computer running the client does not require any specific software installation, except for a compatible modern browser (e.g., Chrome or Firefox).

4.4.7 Rendering time

Once the model is received, the client needs to draw it. In this test, we evaluate how much time is required to draw the views, according to the view models, the number of XY series or time graph rows, and the resolution. Given that getting the model is independent from the drawing operation, the tests in this section use randomly generated models.

XY charts

Several open-source chart libraries are available. They either chose an SVG-based approach or a canvas-based approach. We used Chart.js for the canvas-based library and HighCharts for the SVG-based library. In this test, we compared the rendering time of both techniques. In the first test, we fixed the number of series to 1 and changed the resolution. In the second test, we fixed the resolution to 1000 and changed the number of XY series. Figure 4.16 shows

the results for the first test and Figure 4.17 shows the results for the second.

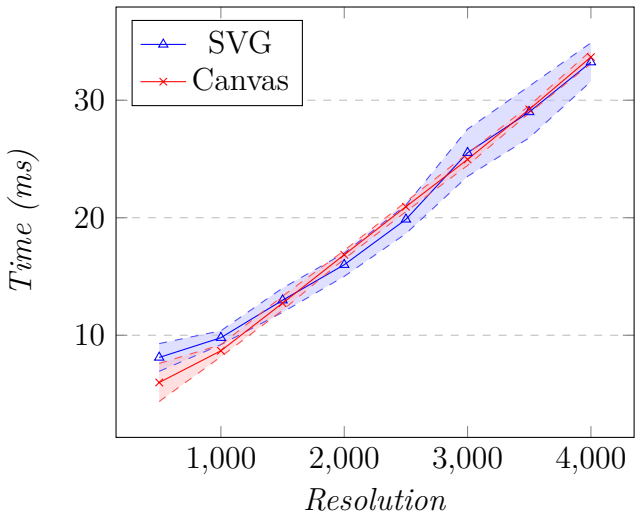

Figure 4.16 Comparison of the rendering time for an XY chart. The number of series is fixed to 1 the resolution varies.

As the number of points increases, the rendering times for SVG and canvas are very close. Less than 10 *ms* are needed to render 500 points and around 35 *ms* for 4000 points.

As the number of XY series increases, we see that the rendering time for SVG is slightly better than canvas. Approximately 300 *ms* are needed to render 35 XY series. However, in practice, it is rare that a user would show that many series on a chart.

Time graph charts

We did not find a time graph chart library that supports nano-scale precision, so we implemented our time graph chart that uses a canvas-based library. The chosen library, called PixiJS, uses hardware acceleration. In the first test, we fixed the number of time graph rows to 25. Given that we generated the model, each time graph row has the same number of states, and we changed this value for the first test. In the second test, we fixed the number of states to 1000, and we changed the number of time graph rows. Table 4.9 shows the results for the first test and table 4.10 shows the results for the second.

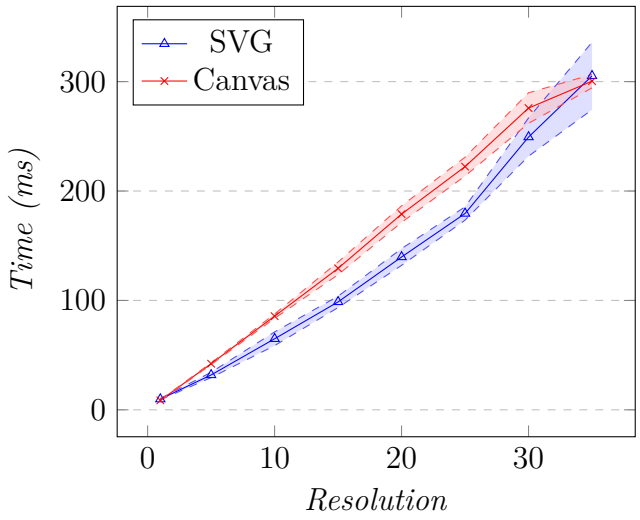

Figure 4.17 Comparison of the rendering time for an XY chart. The resolution is fixed to 1000 and the number of series varies.

Table 4.9 Rendering time for the time graph chart with randomly generated models. The number of time graph rows is fixed to 25.

Resolution (number of states)	Rendering time (ms)
500	6.48
1000	6.13
1500	8.09
2000	13.55
2500	18.70
3000	23.59
3500	29.00
4000	34.85

Table 4.10 Rendering time of the time graph chart with randomly generated models. Fixed resolution to 1000.

Number of time graph row	Rendering time (ms)
25	6.79
50	14.27
75	25.18
100	33.85
125	39.81
150	49.76
175	62.70
200	68.67

4.4.8 Total elapsed time

Finally, in this section, we compare the total elapsed time for showing a view with our client-server solution and with the standalone approach. The elapsed time is the sum of the time to query to Data Provider and to show its results on a view. We used the 2.47 GB trace from the previous section and issued requests covering the whole time interval of the trace. We requested the XY model and the time graph model. The following is a description of the two configurations studied :

— **Standalone** : Locally in Trace Compass, with the implementation of the Data Provider. The Java-based views call the Data Provider explicitly and display the results ;
— **Client-server** : Locally in TraceScape, with Trace Compass as a trace analysis server and the Protobuf serialization format. The JavaScript-based views make HTTP requests to the server and display the results.

XY models

In the first test, we fixed the number of XY series to 1 and changed the resolution. In the second test, we fixed the resolution to 1000 and changed the number of different XY series. Figure 4.18 shows the results of the first test and Figure 4.19 shows the results for the second test.

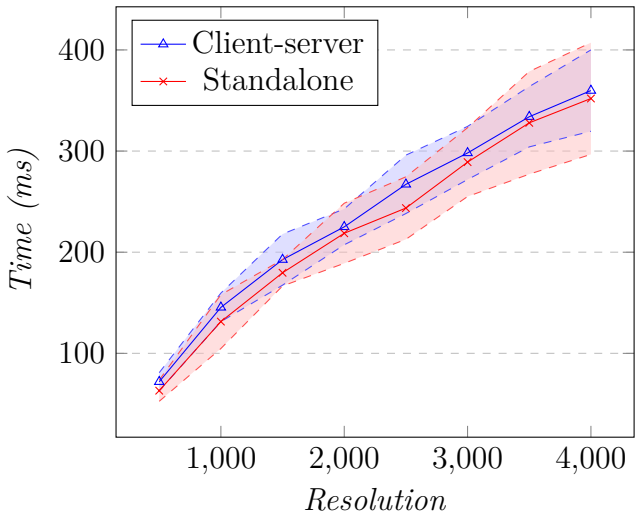

Figure 4.18 Comparison of the total elasped time for the XY model. The number of series is fixed to 1 and the resolution varies.

As the resolution increases, the difference in overhead between the client-server approach and standalone approach is not significant. From the results, the maximum cost is around 25 *ms*

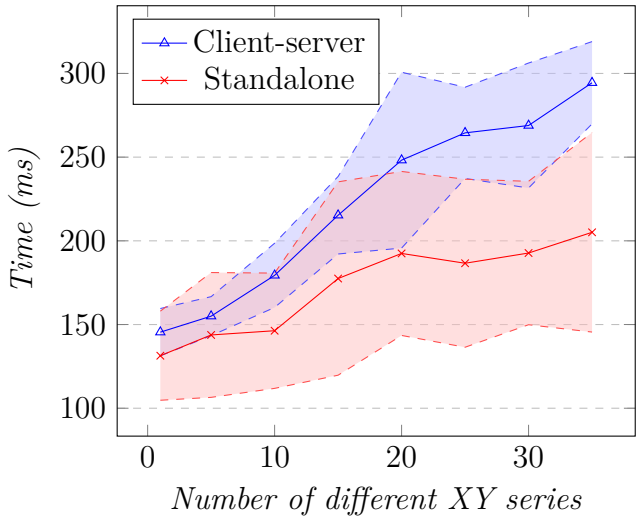

Figure 4.19 Comparison of the total elasped time for the XY model. Fixed resolution to 1000 and changing the number of series.

and the minimum is about 6 *ms*.

However, as the number of XY series increases, the difference in overhead between our solution and the standalone approach is noticeable. This can be explained by the fact that the XY chart library used is SVG-based. For each additional series, 1000 elements are appended in the Document Object Model (DOM), and this impacts the performance significantly. For 35 XY series, the difference is around 90 *ms*.

Time graph models

In the first test, we fixed the number of time graph rows to 25 and changed the resolution. In the second test, we fixed the resolution to 1000 and changed the number of different time graph rows. Figure 4.20 shows the results of the first test and Figure 4.21 shows the results for the second test.

As the resolution increases, the elapsed time for our solution and the standalone approach are close. From the results, the overhead is between 25 *ms* and 65 *ms*, but those values are relatively small compared to the total elapsed time.

However, as the number of time graph rows increases, the difference in elapsed time between the client-server approach and the standalone approach grows slightly. From the results, the overhead is between 30 *ms* and 55 *ms*. Unlike the results in the first test, those values are noticeable compared to the total elapsed time.

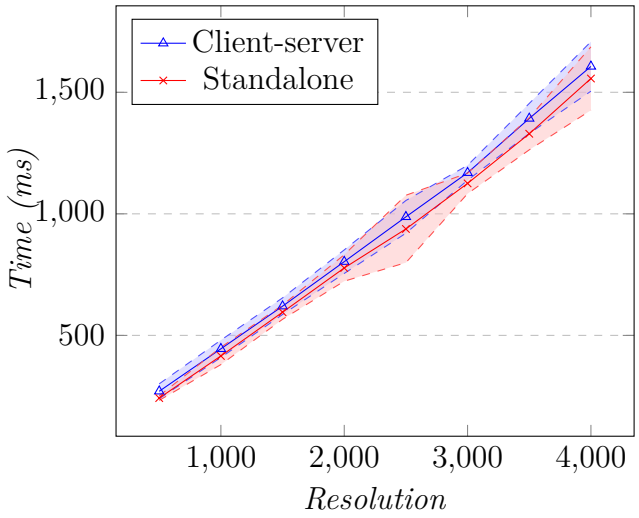

Figure 4.20 Comparison of the total elasped time for the time graph row model. The number of time graph rows is 25 and the resolution varies.

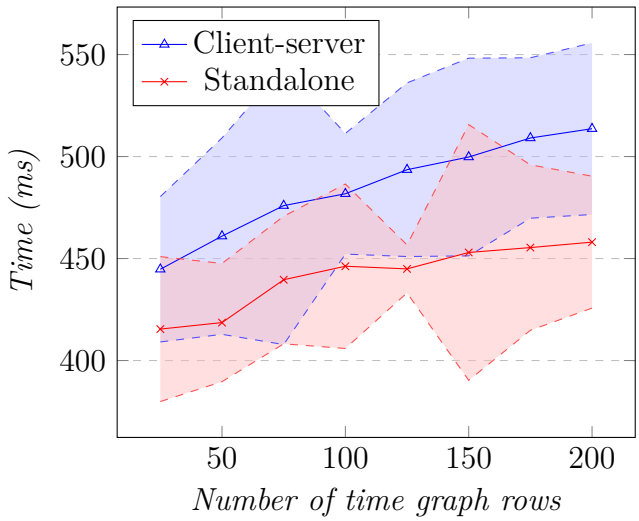

Figure 4.21 Comparison of the total elasped time for the time graph row model. The resolution is fixed to 1000 and the number of time graph rows varies.

4.5 Discussion

In section 4.4, we showed that our solution scales well for large traces, both in terms of data transferred and execution time overhead. While compressed JSON and Protobuf are close when it comes to sending the smallest amount of data over the network, Protobuf has the lowest execution time overhead. Depending on the model requested, the resolution and the number of XY series or time graph rows, using Protobuf, for the worst case, requires less than 700 KB and less than 50 *ms* overhead per request for a trace of 2.47 GB. In comparison with the standalone approach, our proposed solution adds a small and acceptable cost to the total elapsed time.

However, our solution is not optimal for small traces with a size on disk of a few megabytes. Indeed, we could send more data over the network than the trace size. Moreover, we do not know how many requests will be made by the client during a session. For example, with a 1 GB trace, if each request leads to the worst case scenario where we transfer 700 KB per request, and if the client makes more than 1430 requests, we will transfer in total the size of the trace. In practice, it should not happen often. Moreover, losing some time on small traces should not be a concern, as they require very little time to process anyway.

Other communication protocols may be as good or better than HTTP. We investigated RPC-based protocols such as JSON-RPC and gRPC. We exclude the first because we wanted to minimize the amount of data transferred. The second protocol, on the other hand, is using Protobuf as a serialization format and runs over the HTTP/2 protocol (gRPC authors). However, browsers don't fully support the gRPC protocol, and we needed a particular configuration involving a reverse proxy to make it work with the trace analysis server. Hopefully, this limitation will be solved in the future, so we could evaluate precisely how our REST approach compares to an RPC approach.

We proved that our solution is efficient, but we also said that it is flexible, maintainable and reusable. Our focus in this paper was to measure the performance, but it is more difficult to measure and assess those qualities precisely. Because we applied design patterns and software architecture principles, we can intuitively say that our solution has those qualities. However, this intuition is profoundly influenced by the developer work experience. This aspect is another limitation of our work.

4.6 Conclusion and Future Work

In this paper, we presented a client-server architecture for large trace analysis and visualization, that can integrate well with an IDE. By applying the separation of concerns principles,

each component of the architecture has a specific role and responsibility, making them reusable and independently maintainable. We introduced the Trace Analysis Server Protocol (TASP), implemented the API Gateway and the server-side service discovery pattern to ensure that the architecture is flexible and supports multiple heterogeneous clients.

On the server side, instead of sending the state models, we introduced three new architectural components in the trace analysis process to reduce the amount of data transferred. On the client side, we presented some techniques to avoid communication between the client and the server and to optimize the requests to limit the amount of data transferred. To evaluate our solution, we implemented the server architecture in the Trace Compass project and implemented a JavaScript web client. We measured the data transferred per request according to the serialization format, the view model, the number of entries and the resolution. We found that compressed JSON (GZip) and Protobuf are close when it comes to sending the smallest amount of data. However, when we compared the execution time overhead, Protobuf is significantly better than compressed JSON. Then, we showed that our solution scales with large traces. We also discussed why it is not optimal for small traces. Finally, in comparison with the standalone approach, our solution adds a little and acceptable overhead for the total elapsed time.

We believe that our proposed architecture would lead to better integration of tracing and trace analysis in IDEs to better support developers. To our knowledge, this is the first trace visualization tool that addresses the challenges brought forward by the recent modularization of IDEs, with a frontend user interface and backend servers for language parsing, debugging and now tracing. As the systems are becoming more and more complex, we must adapt our tools to detect quickly and effectively defects. Future work could investigate how IDEs could use some components from our architecture, and identify for the developer which part of the code causes problems, using the trace analysis results.

CHAPITRE 5 DISCUSSION GÉNÉRALE

5.1 TraceScape

La figure 5.1 illustre l'interface graphique de notre implémentation de l'architecture proposée du client. Il est possible de voir une vue de type *time graph*, deux vues de type XY et un tableau. À gauche, nous avons l'explorateur de trace, permettant de lister toutes les traces ouvertes sur le serveur.

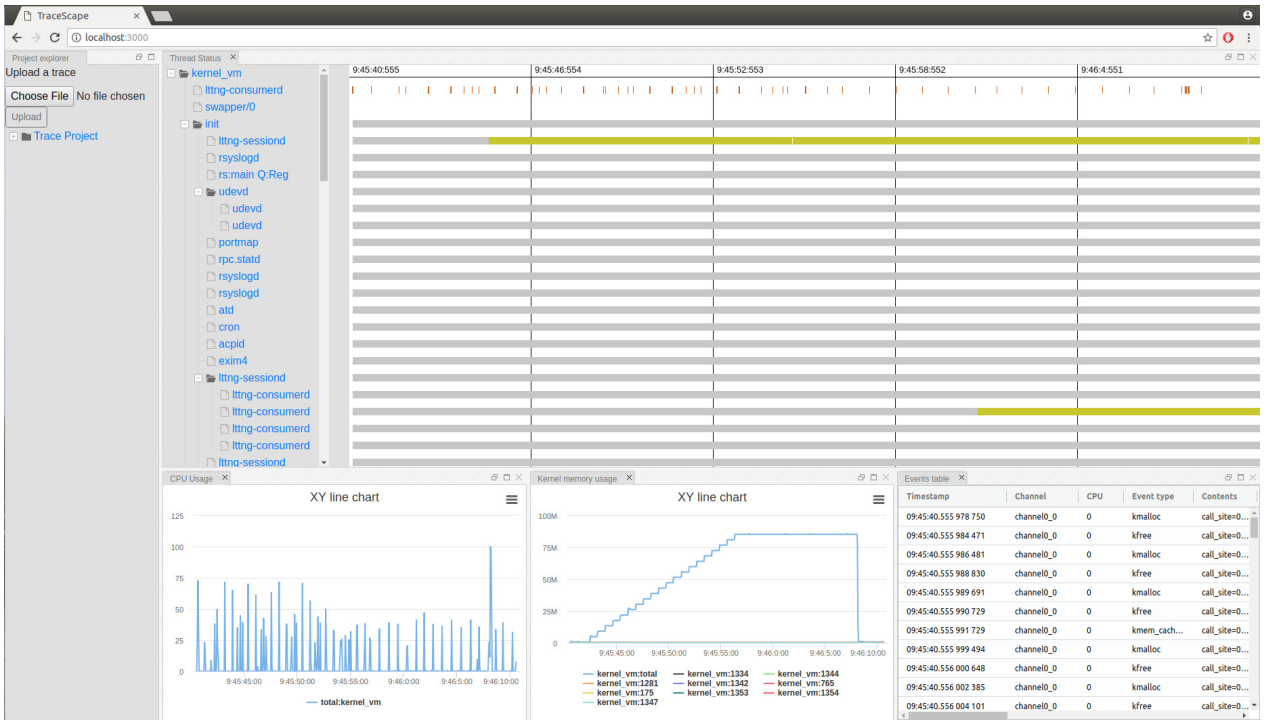

Figure 5.1 Interface graphique du client TraceScape

5.2 Extensibilité

Notre travail s'est arrêté aux modèles de type XY et *time graph*. Néanmoins, il existe d'autres types de vue telles que des tableaux et des diagrammes circulaires que nous n'avons pas eu le temps d'implémenter. Nous proposons dans cette section des modèles qui pourraient être implémentés dans le futur.

5.2.1 Tableaux

Un tableau se compose de métadonnées et de données. Les colonnes représentent les métadonnées et les rangées expriment les données. Dans le cas où le tableau est trop volumineux, le modèle doit supporter le concept de table virtuelle. En effet, si le tableau original contient plusieurs millions de rangées, il serait plus efficace de retourner un modèle qui décrit un sous-tableau, l'index du 1er élément du sous-tableau par rapport au tableau et le nombre de rangées. Nous proposons donc d'avoir les modèles suivants :

Code 5.1 Définition de modèles pour les tableaux

```
/* Contient les metadonnees et les donnees */
struct VirtualTableModel {
    long index;
    long size;
    columns[] TableColumn;
    string[][] rows;
}

/* Decrit la colonne, son identifiant unique et son nom */
struct TableColumn {
    long id;
    string name;
}
```

5.2.2 Diagrammes circulaires

Les diagrammes circulaires ressemblent beaucoup aux diagrammes de type XY. Au lieu d'avoir une collection d'entiers pour décrire l'axe des X, nous proposons d'utiliser une collection d'étiquettes. Chaque étiquette possède un nom et une description plus exhaustive. Nous aurons donc les modèles suivants :

Code 5.2 Définition de modèles pour les diagrammes circulaires

```
struct PieModel {
    string name;
    PieLabel[] labels;
    double[][] values;
}
```

```
struct PieLabel {
    long id;
    string name;
    string description;
}
```

5.3 Retour sur les résultats

Les résultats présentés au chapitre 4 se basent sur des valeurs de test qui peuvent être utilisées en pratique. Les valeurs de résolution s'étendent jusqu'à 4000, ce qui correspond à un écran 4K. Les valeurs du nombre de séries XY s'étendent jusqu'à 35, mais en pratique il est très peu probable qu'un utilisateur veuille afficher 35 séries en même temps sur un écran. D'ailleurs, les résultats obtenus ne s'appliquent que si l'utilisateur affiche un seul type de vue. Si nous avons plusieurs types de vues sur le même écran, le surcoût total serait évidemment plus grand. Puisque nous ne connaissons pas d'avance quelles vues seront utiles à l'utilisateur, nous pouvons avoir une estimation de ce surcoût en additionnant les surcoûts individuels. Supposons que nous avons deux vues de type *time graph* sur un écran 4K disposées de part et d'autre (à gauche et à droite) et que chaque vue affiche 25 *time graph row*. Chaque vue aura donc une résolution maximale de 2000. Pour estimer le surcoût total, nous pouvons additionner les deux surcoûts reliés à une résolution de 2000 pour 25 *time graph row*.

En ce qui concerne le transfert de données, le format Protobuf et le JSON compressé sont nettement plus efficaces que le JSON. Dans certains cas, la compression du JSON est beaucoup plus efficace que le Protobuf, puisque la compression dépend de la nature des données. Par exemple, un tableau contenant uniquement des 0 aura une taille plus petite lorsque GZip est utilisé. Cependant, tel qu'observé dans les résultats au chapitre 4, le surcoût d'utiliser la compression est significativement plus grand que pour les deux autres formats. Par ailleurs, Apache Thrift est souvent comparé à Protobuf, mais certains articles et analyses comparatives démontrent que Protobuf est légèrement plus efficace que Thrift en termes de rapidité de sérialisation et de taille de données produites (Sumaray and Makki, 2012; Maeda, 2012). Bien qu'il aurait été intéressant d'inclure Thrift dans nos comparaisons, tout comme d'autres formats binaires, nos travaux se sont penchés sur l'analyse du surcoût de l'architecture et non de l'analyse du surcoût des formats de sérialisation.

En ce qui concerne la mise à l'échelle, notre solution supporte les traces de grande taille. Le surcoût d'exécution ne varie pas en fonction de la taille de la trace, mais le temps total de chaque requête est plus important. Cela s'explique par le fait que le *State History Tree* (SHT)

contient davantage d'états et le temps de recherche dans cette structure est plus long. Par conséquent, le surcoût relatif devient de plus en plus petit, puisque le serveur d'analyse passe la majeure partie de son temps à chercher dans le SHT. Puisque l'architecture proposée utilise les principes de l'architecture en couche, il sera possible de changer l'implémentation du SHT pour en améliorer les performances et mieux exploiter un environnement infonuagique.

Finalement, en ce qui concerne le surcoût par rapport à l'approche précédente, les résultats avancent un surcoût d'une centaine de millisecondes dans le pire cas. Il est à noter que `TraceScape` n'est qu'un prototype d'implémentation de l'architecture proposée du client et qu'il n'est pas entièrement optimisé. De plus, nous avons choisi de comparer une implémentation en JavaScript avec un client Java.

5.4 Limitation de la solution

Bien que la nouvelle architecture proposée apporte plus de flexibilité que la précédente approche et permette une utilisation distribuée, elle comporte des limitations. Au chapitre 4, nous avons présenté des techniques que le client peut exploiter afin de réduire la quantité de requêtes à faire au serveur et la quantité de données à transférer. Malgré cela, notre solution n'est pas optimale lorsque la taille des traces est petite. Il est possible que l'utilisateur ait à effectuer plusieurs opérations de zoom ou de déplacement durant sa session d'analyse de trace. Conséquemment, il en résulte que la quantité de données nécessaire à transférer est supérieure à la taille de la trace. D'autre part, nous ne connaissons pas d'avance le nombre total de requêtes que le client effectuera au serveur. Par exemple, pour une trace de 1 Mo, si chaque requête nécessite de transférer 100 Ko, il suffit que le client fasse 10 requêtes pour que la taille du transfert dépasse celui de la trace.

En ce qui concerne le TASP, nous avons choisi une approche REST utilisant le protocole HTTP. Nous avons également considéré des protocoles RPC comme JSON-RPC et gRPC pour l'implémentation du TASP. Nous avons exclu le premier à cause du format JSON qui n'est pas optimal en termes de données à transférer. Pour le deuxième, nous nous sommes heurtés à des limitations. Le cadriciel n'est pas nativement supporté par les navigateurs web et un projet sur Github [1], externe au projet original, le permet. Nous avions réussi à faire fonctionner `TraceScape` avec gRPC, mais cela impliquait de mettre en place une configuration spéciale :

— Un certificat SSL auto signé. Les navigateurs web ne supportent pas le protocole HTTP/2 si les communications ne sont pas cryptées [2]. Si nous n'utilisons pas HTTP/2,

1. https ://github.com/grpc/grpc-web
2. https ://http2.github.io/faq/#does-http2-require-encryption

gRPC utilise par défaut HTTP/1.1 qui n'est pas binaire.

— Un proxy inverse. Le navigateur web ne supporte pas entièrement le protocole gRPC et il faut mettre en place un proxy inverse qui répond à des requêtes HTTP standard et réachemine les requêtes vers un serveur gRPC.

Toutes ces configurations ne nous permettent pas de profiter pleinement des performances de gRPC, puisqu'elles introduisent des surcoûts. En conséquence, notre implémentation du TASP s'est limitée à l'utilisation de HTTP. Il s'agit d'une autre limitation de notre travail. Les résultats obtenus au chapitre 4 testent le surcoût de l'architecture dans sa globalité. L'implémentation du TASP n'est pas nécessairement la plus optimale en termes de temps d'exécution et laisse place à des optimisations. Par exemple, `Monto` a choisi la librairie *ZeroMQ* et `Kómpos` a choisi les *Web Sockets*.

CHAPITRE 6 CONCLUSION

Afin de conclure ce mémoire, nous présentons une synthèse des contributions de nos travaux de recherche. Finalement, nous énoncerons des propositions d'améliorations pour les travaux futurs.

6.1 Synthèse des travaux

Dans ces travaux, nous avons proposé une architecture client-serveur combinée avec une approche d'architecture orientée service pour la conception d'un serveur d'analyse de trace. Ce modèle permettrait donc à n'importe quel EDI d'interroger le serveur pour les résultats d'analyse de trace par le biais d'un protocole de communication.

Dans un premier temps, nous avons extrait des modèles génériques représentant les résultats d'analyse. Nos travaux se sont limités aux modèles pour les vues de type XY et les vues de type time graph, ce qui couvre les types de vues les plus importants des outils de visualisation de trace. Ensuite, nous avons introduit une nouvelle couche d'abstraction dans Trace Compass, les Data Providers, responsables de calculer le modèle selon l'intervalle de temps voulu, la résolution et les routines. Par la suite, nous avons implémenté une couche de services, responsable de répondre à des requêtes HTTP, afin de supporter plusieurs clients hétérogènes. L'architecture choisie pour le serveur d'analyse de trace utilise également les principes de l'architecture en couche, ce qui facilite le changement d'implémentation de chaque couche sans avoir d'impact sur les autres couches.

Considérant que le client pourrait recevoir des informations d'analyse de trace de plusieurs serveurs, nous avons introduit le Trace Analysis Server Protocol (TASP), un protocole de communication basé sur HTTP. Le TASP est conçu sur les principes architecturaux du Representational State Transfer (REST) et le format d'échange choisi est le Protocol Buffer, un format binaire compact. L'introduction d'un protocole commun pour tous les serveurs d'analyse de trace permet d'avoir un faible couplage, et l'effort pour l'ajout ou le changement d'implémentation d'un serveur est minime.

Finalement, nous avons implémenté un client web, TraceScape, qui supporte le TASP et communique avec le serveur d'analyse de trace Trace Compass. Des tests de performance ont été menés et révèlent un surcoût acceptable. Nous concluons donc que les quatre objectifs de recherches formulés au chapitre 1 ont été atteints.

6.2 Améliorations futures

La nouvelle architecture proposée laisse place à de nouvelles possibilités et améliorations dans le cadre de ce projet de recherche. Puisque le TASP introduit un faible couplage entre le client et le serveur, ces deux composantes peuvent être développées et optimisées indépendamment. De plus, le TASP peut évoluer et être optimisé sans avoir un impact majeur sur le client et le serveur.

En ce qui concerne le serveur, les améliorations futures concernent l'architecture des analyses, du gestionnaire d'état et du State History Tree. À l'heure actuelle, le SHT est une structure de données écrite sur disque, mais cette solution n'est probablement pas optimale et viable dans un environnement infonuagique multimachines et multiutilisateurs simultanés. Une première solution serait simplement d'utiliser un système de fichiers distribués pour le stockage du SHT. Cependant, cette approche a ses limites et ce système de fichiers deviendrait rapidement un goulot d'étranglement. Par conséquent, il serait intéressant que de futurs travaux s'intéressent à adapter cette structure pour qu'elle soit utilisable avec des outils comme Cassandra ou Hadoop.

En ce qui concerne le client, les travaux futurs concernent l'amélioration des performances et son intégration aux EDI. La conception et l'implémentation du client TraceScape ont été faites de façon à faciliter son intégration dans des EDI comme VS Code ou Theia. Cependant, nos recherches se sont arrêtées avant cette étape et de futurs travaux pourraient s'y attarder plus amplement. Par ailleurs, l'implémentation de TraceScape n'est pas nécessairement la plus optimale, puisqu'elle utilise des librairies externes pour la visualisation.

Finalement, en ce qui concerne le protocole, les améliorations futures concernent son évolution. La définition du protocole n'est pas finale et plusieurs aspects n'ont pas été couverts. Par exemple, la présentation des modèles n'a pas été spécifiée par le TASP. En effet, le protocole devrait fournir un mécanisme pour donner au client des indices sur comment afficher le modèle : couleur, style, épaisseur, etc. Ces indices ne sont pas obligatoires et peuvent être outrepassés par le client.

RÉFÉRENCES

P. Barham, R. Isaacs, R. Mortier, et D. Narayanan, “Magpie : Online modelling and performance-aware systems”, dans *9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*. USENIX, May 2003, pp. 85–90.

T. Bird, “Measuring function duration with ftrace”, dans *Proceedings of the Linux Symposium*, 2009, pp. 47–54.

D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, et D. Winer, “Simple object access protocol (soap) 1.1”, 2000. En ligne : `https://www.w3.org/TR/2000/NOTE-SOAP-20000508/`

T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, et F. Yergeau, “Extensible markup language (xml).” *World Wide Web Journal*, vol. 2, no. 4, pp. 27–66, 1997.

A. Chan, W. Gropp, et E. Lusk, “An efficient format for nearly constant-time access to arbitrary time intervals in large trace files”, *Scientific Programming*, vol. 16, no. 2-3, pp. 155–165, 2008.

G. Coulouris, J. Dollimore, T. Kindberg, et G. Blair, *Distributed Systems : Concepts and Design*, 5e éd. USA : Addison-Wesley Publishing Company, 2011.

M. Desnoyers, “Common trace format (ctf) specification (v1. 8.2)”, *Common Trace Format GIT repository*, 2012. En ligne : `https://diamon.org/ctf/`

M. Desnoyers et M. Dagenais, “Os tracing for hardware, driver and binary reverse engineering in linux”, dans *Linux CodeBreakers Journal*, vol. 1, no. 2, 2006.

M. Desnoyers et M. R Dagenais, “The lttng tracer : A low impact performance and behavior monitor for gnu/linux”, dans *Ottawa Linux Symposium*, January 2006.

N. Duca et D. Sinclair. Trace event format. Visité le 2017-01-23. En ligne : `https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview`

J. Edge. (2009) Perfcounters added to the mainline. Visité le 2018-02-21. En ligne : `https://lwn.net/Articles/339361/`

S. Efftinge et A. Kosyakov, “Theia - one ide framework for desktop & cloud”, communication présentée à EclipseCon France 2017, Toulouse, France, June 21-22 2017.

Ericsson. Trace analysis server protocol. En ligne : https://theia-ide.github.io/trace-server-protocol

D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, et F. Wolf, "Open trace format 2 : The next generation of scalable trace formats and support libraries." dans *PARCO*, vol. 22, 2011, pp. 481–490.

N. Ezzati-Jivan et M. R. Dagenais, "Multi-scale navigation of large trace data : A survey", *Concurrency and Computation : Practice and Experience*, vol. 29, no. 10, 2017. DOI : 10.1002/cpe.4068. En ligne : https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4068

R. T. Fielding, "Architectural styles and the design of network-based software architectures", Thèse de doctorat, University of California, Irvine, 2000. En ligne : https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf

E. Gamma, R. Helm, R. Johnson, et J. Vlissides, *Design patterns : elements of reusable object-oriented software*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995.

D. Garlan et M. Shaw, "An introduction to software architecture", dans *Advances in software engineering and knowledge engineering*. World Scientific, 1993, pp. 1–39.

M. Gebai et M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux : Design, implementation, and overhead", *ACM Computing Surveys (CSUR)*, vol. 51, no. February, 2018. DOI : 10.1145/3158644. En ligne : http://doi.acm.org/10.1145/3158644

Google. (2018) Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data. En ligne : https://developers.google.com/protocol-buffers/

B. Gregg. (2014) strace wow much syscall. En ligne : http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html

B. Gregg et J. Mauro, *DTrace : Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*, 1er éd. Upper Saddle River, NJ, USA : Prentice Hall Press, 2011.

J.-R. W. Group *et al.*, "Json-rpc 2.0 specification", January 2013. En ligne : https://www.jsonrpc.org/specification

W. W. S. A. W. Group *et al.*, “Web services architecture”, *World Wide Web Consortium*, 2004. En ligne : `https://www.w3.org/TR/ws-arch/`

T. gRPC authors. About grpc. En ligne : `https://grpc.io/about/`

M. Hericko, M. B. Juric, I. Rozman, S. Beloglavec, et A. Zivkovic, “Object serialization analysis and comparison in java and. net”, *ACM Sigplan Notices*, vol. 38, no. 8, pp. 44–54, 2003.

A. Homer, J. Sharp, L. Brader, M. Narumoto, et T. Swanson, “Cloud design patterns”, *Microsoft*, 2014.

E. International, “The json data interchange syntax”, *ECMA-404*, 2017.

S. Keidel, W. Pfeiffer, et S. Erdweg, “The ide portability problem and its solution in monto”, dans *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2016, pp. 152–162.

A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony *et al.*, “Score-p : A joint performance measurement runtime infrastructure for periscope, scalasca, tau, and vampir”, dans *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.

K. G. Kouamé, “Langage dédié et analyse automatisée pour la détection de patrons au sein de traces d’exécution”, Mémoire de maîtrise, École Polytechnique de Montréal, Montréal, QC, 2015. En ligne : `https://publications.polymtl.ca/1900/`

J. Kraft, A. Wall, et H. Kienle, “Trace recording for embedded systems : Lessons learned from five industrial projects”, dans *International Conference on Runtime Verification*. Springer, 2010, pp. 315–329.

C. Larman, *Applying UML and Patterns : An Introduction to Object Oriented Analysis and Design and Interative Development*. Pearson Education, 2012.

K. Maeda, “Performance evaluation of object serialization libraries in xml, json and binary formats”, dans *Digital Information and Communication Technology and it’s Applications (DICTAP), 2012 Second International Conference on*. IEEE, 2012, pp. 177–182.

S. Marr, C. Torres Lopez, D. Aumayr, E. Gonzalez Boix, et H. Mössenböck, “A concurrency-agnostic protocol for multi-paradigm concurrent debugging tools”, dans *ACM SIGPLAN Notices*, vol. 52, no. 11. ACM, 2017, pp. 3–14.

M. Masse, *REST API Design Rulebook : Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, Inc., October 2011.

Microsoft. Visual studio code. En ligne : `https://github.com/Microsoft/vscode`

I. MongoDB, "Mongodb architecture guide", Rapp. tech., 2018. En ligne : `https://www.mongodb.com/collateral/mongodb-architecture-guide`

A. Montplaisir, N. Ezzati-Jivan, F. Wininger, et M. Dagenais, "Efficient model to query and visualize the system states extracted from trace data", dans *Runtime Verification*, Springer. Springer Berlin Heidelberg, 2013, pp. 219–234.

I. Park et R. Buch, "Event tracing : Improve debugging and performance tuning with etw", p. 81, April 2007.

V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, et J. Chen, "Locating system problems using dynamic instrumentation", dans *2005 Ottawa Linux Symposium*, 2005, pp. 49–64.

L. Prieur-Drevon, R. Beamonte, et M. Dagenais, "R-sht : A state history tree with r-tree properties for analysis and visualization of highly parallel system traces", *Journal of Systems and Software*, vol. 135, pp. 55–68, 2018. DOI : `https://doi.org/10.1016/j.jss.2017.09.023`. En ligne : `http://www.sciencedirect.com/science/article/pii/S0164121217302108`

L. Prieur-Drevon, "Structures de données hautement extensibles pour le stockage sur disque de séries temporelles hétérogènes", Mémoire de maîtrise, École Polytechnique de Montréal, Montréal, QC, 2017. En ligne : `https://publications.polymtl.ca/2489/`

L. Prieur-Drevon, R. Beamonte, N. Ezzati-Jivan, et M. R. Dagenais, "Enhanced state history tree (esht) : a stateful data structure for analysis of highly parallel system traces", dans *Big Data (BigData Congress), 2016 IEEE International Congress on*. IEEE, 2016, pp. 83–90.

T. L. project. Lttng scope. En ligne : `https://github.com/lttng/lttng-scope`

A. Prunicki, "Apache thrift", june 2009. En ligne : `https://objectcomputing.com/resources/publications/sett/june-2009-apache-thrift`

C. Richardson, "Pattern : Api gateway", *Backend for Front-End*, pp. 37–40, 2017.

——, "Pattern : Server-side service discovery", *Backend for Front-End*, 2017.

S. Rostedt, “Finding origins of latencies using ftrace”, dans *11th Real-Time Linux Workshop*, 2009, pp. 28–30.

M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, et S. Cranford, “Open| speedshop : An open source infrastructure for parallel performance analysis”, *Scientific Programming*, vol. 16, no. 2-3, pp. 105–121, 2008. DOI : 10.1155/2008/713705. En ligne : http://dx.doi.org/10.1155/2008/713705

J. Siegel, “Omg overview : Corba and the oma in enterprise computing”, *Communications of the ACM*, vol. 41, no. 10, pp. 37–43, 1998. DOI : 10.1145/286238.286246. En ligne : http://doi.acm.org/10.1145/286238.286246

B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, et C. Shanbhag, “Dapper, a large-scale distributed systems tracing infrastructure”, Google, Inc, Rapp. tech., 2010. En ligne : https://research.google.com/archive/papers/dapper-2010-1.pdf

M. Slee, A. Agarwal, et M. Kwiatkowski, “Thrift : Scalable cross-language services implementation”, *Facebook White Paper*, vol. 5, no. 8, 2007.

H. Strubel, “Monitoring distributed traces with kieker”, Mémoire de maîtrise, Institut für Informatik, Kiel, Allemagne, 2017. En ligne : http://oceanrep.geomar.de/38734/

A. Sumaray et S. K. Makki, “A comparison of data serialization formats for optimal efficiency on a mobile platform”, dans *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, série ICUIMC ’12. ACM, 2012, pp. 48 :1–48 :6. DOI : 10.1145/2184751.2184810. En ligne : http://doi.acm.org/10.1145/2184751.2184810

A. S. Tanenbaum et M. v. Steen, *Distributed Systems : Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 2006.

U. Technologies. Jaeger - a distributed tracing system. En ligne : https://github.com/jaegertracing/jaeger

M. H. Valipour, B. AmirZafari, K. N. Maleki, et N. Daneshpour, “A brief survey of software architecture concepts and service oriented architecture”, dans *2009 2nd IEEE International Conference on Computer Science and Information Technology*. IEEE, Aug 2009, pp. 34–38. DOI : 10.1109/ICCSIT.2009.5235004

J. Waldo, “Remote procedure calls and java remote method invocation”, *IEEE concurrency*, vol. 6, no. 3, pp. 5–7, July 1998. DOI : 10.1109/4434.708248

J. C. Wileden et A. Kaplan, “Software interoperability : Principles and practice”, dans *Proceedings of the 21st International Conference on Software Engineering*, série ICSE ’99. New York, NY, USA : ACM, 1999, pp. 675–676. DOI : 10.1145/302405.302952. En ligne : http://doi.acm.org/10.1145/302405.302952

F. Wininger, “Conception flexible d’analyses issues d’une trace système”, Mémoire de maîtrise, École Polytechnique de Montréal, Montréal, QC, 2014. En ligne : https://publications.polymtl.ca/1359/

F. Wininger, N. Ezzati-Jivan, et M. R. Dagenais, “A declarative framework for stateful analysis of execution traces”, *Software Quality Journal*, vol. 25, no. 1, pp. 201–229, Mar 2017. DOI : 10.1007/s11219-016-9311-0. En ligne : https://doi.org/10.1007/s11219-016-9311-0

C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, et W. Gropp, “From trace generation to visualization : A performance framework for distributed parallel systems”, dans *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2000, p. 50.

O. Zaki, E. Lusk, W. Gropp, et D. Swider, “Toward scalable performance visualization with jumpshot”, *The International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 277–288, 1999.

ANNEXE A SPÉCIFICATION DU TRACE ANALYSIS SERVER PROTOCOL

Le Trace Analysis Server Protocol (TASP) est le résultat d'une collaboration entre l'équipe d'EfficiOS travaillant sur LTTng Scope, l'équipe d'Ericsson travaillant sur Trace Compass et le laboratoire DORSAL. Cette annexe est une brève introduction et explication des principaux mécanismes du TASP. La spécification la plus exhaustive et la plus récente du protocole est disponible sur `https://theia-ide.github.io/trace-server-protocol`.

A.1 Architecture de type REST

Le TASP est basé sur le style architectural *Representational state transfer* (REST). Le client accède et manipule des ressources par l'intermédiaire d'un *Uniform Resource Identifier* (URI). Un même URI peut servir à effectuer plusieurs actions sur une ressource. Pour ce faire, la requête HTTP doit spécifier la méthode à utiliser : `GET`, `POST`, `DELETE` ou `PUT`. En ce qui concerne le TASP, la méthode `GET` est strictement utilisée pour l'obtention d'une ressource. Les autres méthodes sont respectivement utilisées pour l'ajout, la suppression et la mise à jour des ressources.

Les segments de l'URI au pluriel signifient qu'une collection est manipulée. Par exemple, pour la gestion des traces, une requête `GET` sur la route `/traces` retourne une collection de ressources représentant une trace. Chaque ressource d'une collection possède un identifiant unique et le TASP permet d'obtenir des informations sur des ressources spécifiques de la collection. Ainsi, pour obtenir l'information sur une trace spécifique dans la collection, il suffit d'invoquer une requête `GET` sur la route `/traces/{traceId}` où la section `traceId` doit être remplacée par l'identifiant de la trace voulue.

A.2 Gestion des erreurs

Toute requête réussie retourne une réponse HTTP avec le code `200`. Autrement, le serveur de trace retourne une réponse HTTP contenant un code d'erreur. Lorsqu'il s'agit d'une erreur provenant du serveur, une erreur de type `5xx` est lancée. Lorsque le client effectue une requête avec des paramètres incorrects, une erreur de type `4xx` est retournée. Chaque erreur est accompagnée d'une description détaillant les causes de l'erreur. Il est de la responsabilité du client de traiter et gérer les erreurs.

A.3 Gestion des traces et collections de traces

Certaines analyses peuvent s'effectuer sur plusieurs traces. Le protocole fait donc une distinction entre une trace et une collection de trace, appelée *experiment*. Le TASP supporte l'obtention des modèles de type XY ou *time graph* à partir d'une trace ou d'un *experiment*. Le tableau A.1 décrit la liste des routes pour la gestion des traces et des *experiments*. Seuls les routes utilisant les méthodes `PUT` et `POST` requièrent des paramètres de type `FormData`. Ils seront détaillés plus bas.

Tableau A.1 Routes pour la gestion des traces et des *experiments*

Méthode	Route	Description
GET	/traces	Retourne la liste des traces analysées
GET	/traces/{traceId}	Retourne les détails sur une trace spécifique
POST	/traces	Ajoute une trace dans la collection des traces à analyser
DELETE	/traces/{traceId}	Retire une trace spécifique de la collection des traces analysées
GET	/experiments	Retourne la liste des *experiments*
POST	/experiments	Créer un nouvel *experiment*
GET	/experiments/{expID}	Retourne les détails sur un *experiment* spécifique
PUT	/experiments/{expID}	Met à jour le contenu d'un *experiment* (traces)
DELETE	/experiments/{expID}	Retire un *experiment* spécifique de la collection des *experiments*

1. **POST** /traces
   - path (obligatoire) : l'URL qui définit l'emplacement de la trace
   - name : le nom de la trace sur le serveur
   - typeID : le format de la trace (CTF, OTF, etc.)
2. **POST** /experiments
   - name (obligatoire) : le nom de *l'experiment*
   - traces (obligatoire) : la liste des identifiants de traces qui composeront *l'experiment*
3. **PUT** /experiments/{expID}
   - name (obligatoire) : le nouveau nom de *l'experiment*
   - traces (obligatoire) : la nouvelle liste des identifiants de traces qui composeront *l'experiment*

A.4 Obtention des modèles de type XY

L'obtention des modèles XY se décompose en deux possibles routes. L'une sert à obtenir la liste des series et l'autre sert à obtenir les coordonnées du graphique. Puisque nous pouvons obtenir les modèles XY à partir d'une trace ou d'un *experiment*, nous avons donc quatre routes différentes. Le tableau A.2 décrit les routes pour l'obtention des modèles de type XY. Chaque analyse de trace est associée à un identifiant unique et cet identifiant doit être spécifié dans la route. Par exemple, si nous voulons les résultats d'analyse d'utilisation CPU, une requête sur la route `/traces/ma-trace/outputs/cpu/xy` permet de retourner les coordonnées du graphique. Ce mécanisme nous permet de supporter d'autres analyses dans le futur sans apporter de modifications au protocole. Le TASP spécifie également une route spéciale qui retourne la liste des identifiants d'analyse qui sont supportées par le serveur.

Tableau A.2 Routes pour l'obtention des modèles de type XY

Méthode	Route	Description
GET	/traces/{traceId}/outputs/{outputID}/tree	Retourne la liste des séries
GET	/traces/{traceId}/outputs/{outputID}/xy	Retourne la liste des coordonnées du graphique
GET	/experiments/{expID}/outputs/{outputID}/tree	Retourne la liste des séries
GET	/experiments/{expID}/outputs/{outputID}/xy	Retourne la liste des coordonnées du graphique

Les paramètres de type `Query Param` des routes pour l'obtention des séries sont :

- low : l'index de la première entrée voulue
- size : le nombre d'entrées voulues
- start : le temps de début
- end : le temps de fin

Les paramètres obligatoires de type `Query Param` des routes pour l'obtention des coordoonnées sont :

- start (obligatoire) : le temps de début
- end (obligatoire) : le temps de fin
- count (obligatoire) : la résolution (nombre de points)
- series (obligatoire) : la liste des séries voulues

A.5 Obtention des modèles de type *time graph*

Tout comme pour l'obtention des modèles de type XY, il faut spécifier l'identifiant de l'analyse pour obtenir les résultats d'une analyse. En plus des routes pour l'obtention des entrées, des

états, le TASP ajoute une route supplémentaire pour les modèles de type *time graph* pour requérir les *time graph arrows*.

Tableau A.3 Routes pour l'obtention des modèles de type *time graph*

Méthode	Route	Description
GET	/traces/{traceId}/outputs/{outputID}/tree	Retourne la liste des séries
GET	/traces/{traceId}/outputs/{outputID}/states	Retourne la liste des coordonnées du graphique
GET	/traces/{traceId}/outputs/{outputID}/arrows	Retourne la liste des coordonnées du graphique
GET	/experiments/{expID}/outputs/{outputID}/tree	Retourne la liste des séries
GET	/experiments/{expID}/outputs/{outputID}/states	Retourne la liste des coordonnées du graphique
GET	/experiments/{expID}/outputs/{outputID}/arrows	Retourne la liste des coordonnées du graphique

Les paramètres de type `Query Param` des routes pour l'obtention des entrées sont :

- low : l'index de la première entrée voulue
- size : le nombre d'entrées voulues
- start : le temps de début
- end : le temps de fin

Les paramètres obligatoires de type `Query Param` des routes pour l'obtention des états sont :

- start (obligatoire) : le temps de début
- end (obligatoire) : le temps de fin
- count (obligatoire) : la résolution (nombre de points)
- entries (obligatoire) : la liste des entrées voulues

Les paramètres obligatoires de type `Query Param` des routes pour l'obtention des *time graph arrows* sont :

- start (obligatoire) : le temps de début
- end (obligatoire) : le temps de fin
- count (obligatoire) : la résolution (nombre de points)